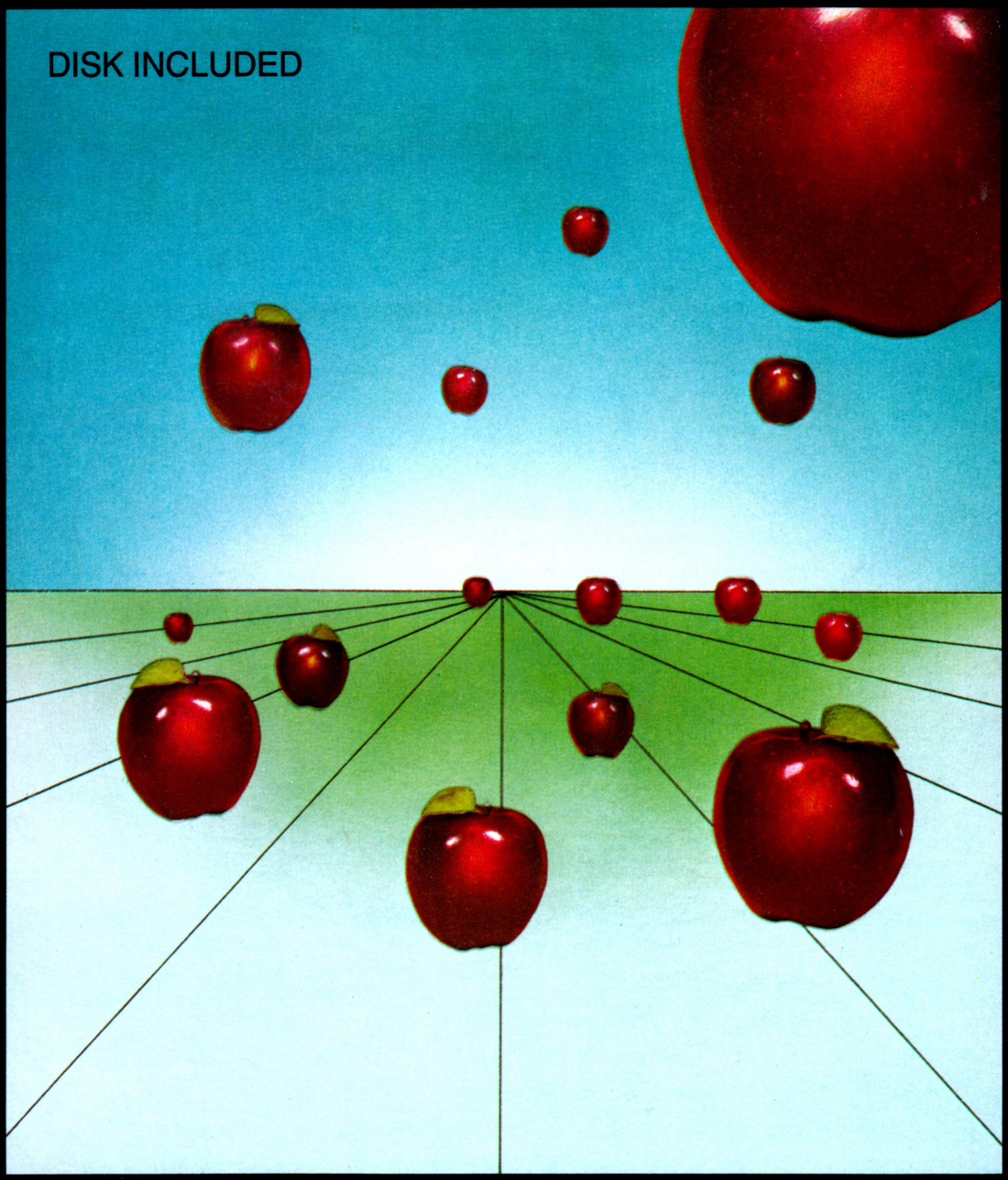
Apple II-6502

A SPECTRUM BOOK

ASSEMBLY LANGUAGE TUTOR



A COMPLETE BOOK - SOFTWARE TUTORIAL - MONITOR PROGRAM FOR 3.3 DOS INCLUDED - HIGH RESOLUTION/LOW RESOLUTION GRAPHICS APPLICATIONS - APPLE II INTERFACING APPLICATIONS

RICHARD HASKELL

TUTOR DISKETTE/REGISTRATION CARD ENCLOSED

RICHARD HASKELL

APPLE II
6502 Assembly Language Tutor

PRENTICE-HALL, INC., Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging in Publication Data

Haskell, Richard E.
Apple II-6502 assembly language tutor.

"A Spectrum Book."

Includes index.

1. Apple II (Computer) — Programming. 2. 6502 (Computer) — Programming. 3. Assembler language (Computer program language) I. Title. II. Title: Apple 2-6502 assembly language tutor. III. Title: Apple two-6502 assembly language tutor

QA76.8.A662H275 1983 001.64'2 82-21459
ISBN 0-13-039230-8 (pbk.)

This book is available at a special discount when ordered in bulk quantities. Contact Prentice-Hall, Inc., General Publishing Division, Special Sales, Englewood Cliffs, N.J. 07632.

A Spectrum Book. Printed in the United States of America.

10 9 8 7 6 5 4 3

ISBN 0-13-039230-8 {PBK.}

Editorial/production supervision by Rita Young
Cover design ©1983 by Jeannette Jacobs
Cover illustration by John Brandes
Manufacturing buyer: Trudy Pisciotti

Prentice-Hall International, Inc., *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Whitehall Books Limited, *Wellington, New Zealand*
Editora Prentice-Hall do Brasil Ltda., *Rio de Janeiro*

Contents

Preface v

1
Getting Started 1

2
The 6502 Microprocessor 3

3
Computer Memory 17

4
The 6502 Registers 27

5
6502 Arithmetic 39

6
Branching Instructions 51

7
The Stack and Subroutines 57

8
Addressing Modes 73

9
Displaying Characters on the Screen 86

10
Low-Resolution Graphics 106

11
High-Resolution Graphics 118

12
Using the Game I/O Connector 134

13
Using the Peripheral I/O Slots 141

14
The 6821 Peripheral Interface Adapter (PIA) 160

15
Interrupts 173

16
Serial I/O—The ACIA 182

Appendices 195

Index 231

Preface

To get the most out of an Apple II computer it is sometimes necessary to write programs in assembly language. To learn how the Apple II works and how to interface it to the outside world it is necessary to understand the 6502 microprocessor. This book is designed to help you understand the 6502 microprocessor by using a special TUTOR monitor that will allow you to easily see what is going on inside the computer.

You will get the most out of this book if you know how to program the Apple II in BASIC. A companion book entitled *Apple BASIC* (Prentice-Hall, 1982) will provide you with the necessary background.

The strategy of this book is "learning by doing." The TUTOR monitor will allow you to write simple machine language programs and watch what happens to registers and memory locations as you single step through a program. In this way you can see exactly what is going on.

After some introductory remarks in Chapter 1 you will begin to use the TUTOR monitor in Chapter 2. You will learn how hexadecimal and binary numbers are related to decimal numbers.

In Chapter 3 you will learn how to examine and change the contents of any memory location. You will see how ASCII codes are used to represent characters in the computer.

The 6502 registers are described in Chapter 4. Instructions associated with the accumulator, the index registers, the stack pointer, and the program counter are illustrated. Binary and decimal arithmetic are described in Chapter 5. The use of the condition codes in branching instructions is discussed in Chapter 6.

You will learn about the stack and subroutines in Chapter 7. In this chapter you will also learn how to use the Apple II speaker and keyboard. The important topic of addressing modes is covered in Chapter 8. All 6502 addressing modes are described with simple examples to illustrate their use.

The way in which the Apple II displays characters on the screen is described in Chapter 9. Low-resolution graphics are discussed in Chapter 10, and high-resolution graphics are covered in Chapter 11.

The remaining chapters in the book are concerned with interfacing the Apple II to the outside world. Chapter 12 shows how to interface a six-channel A/D converter through the game I/O connector. The use of the Apple II peripheral I/O slots is discussed in Chapter 13. The 6821 peripheral interface adapter (PIA) is described in Chapter 14, where its use in a parallel printer interface is illustrated.

The important topic of interrupts is covered in Chapter 15, in which a program for a real-time clock is given. Serial communication with an Apple II is discussed in Chapter 16, which shows how to use an asynchronous communication interface adapter (ACIA) to turn the Apple II into a computer terminal.

Many colleagues and students have influenced the development of this book. Particular assistance was received from Glenn Jackson, Thomas Windeknecht, and Osman Altan. Their stimulating discussions, probing questions, and critical comments are greatly appreciated. Special thanks go to Sharon Rix, whose typing skill made the preparation of the manuscript a pleasure.

Acknowledgment is made to the following for granting permission to reprint materials:

Figures 3.6, 7.15, 9.3, 9.8, 11.4, 12.1 (chart), 13.2, and table 13.1 are reprinted from the *Apple II Reference Manual.* Copyright ©1981 by Apple Computer, Inc. Used with permission from Apple Computer, Inc.

Figures 12.8, 14.1, 16.2, and Data Sheets in Appendix D (pp. 206–30) are reprinted Courtesy of Motorola, Inc.

1

Getting Started

ABOUT THE BOOK

How do microprocessors and microcomputers work, and how can you use them for your own applications? That is what this book is about. Although you will learn how to program in assembly language, this is not just another book on assembly language programming. You can learn assembly language programming by using a cross-assembler on a large main-frame computer. But this book will also show you how microprocessors and microcomputers work.

This book uses a special monitor program called TUTOR that runs on an Apple II microcomputer to help you learn about microprocessors. The TUTOR monitor shows you what is going on inside the computer at any instant of time. It has been specially designed to be easy to use and to make it easier for you to learn about microprocessors.

If you write long assembly language programs (you should want to after reading this book), you will want to use an assembler that automatically converts instruction mnemonics to machine language code. However, you will gain considerable insight into the operation of microprocessors if you initially do this conversion by hand for some short programs. This is the procedure we will follow in this book. The TUTOR monitor will make it easy for you to enter your programs into the computer, run them, and debug them.

A summary about the TUTOR monitor and a description of how to run it are given in Appendix B. You should take a quick look at this ap-

pendix now. It will be a helpful guide to you until you become familiar with the use of the TUTOR monitor. Most of the commands will be introduced in the book, with examples, as they are needed. You will find the TUTOR monitor very easy to use.

With all of the high-level languages around, why should you be interested in learning assembly language? There are several reasons. First, assembly language programs are fast. It is not uncommon for an assembly language program to execute orders of magnitude faster than a corresponding program written in BASIC. This can be important if you want to fill the screen instantaneously with information, or if you want to control some time-critical process.

Second, assembly language programming lets you access the lowest levels of the computer hardware. This is essential when interfacing the computer to external devices or when trying to get the most performance from the computer hardware. Third, most applications of microprocessors are in dedicated systems in which a single program is always executed. This program is normally stored as machine code in a read only memory (ROM) or in a programmable read only memory (PROM). Most such programs are written in assembly language.

Finally, learning assembly language will give you a better understanding of how a computer works. There has traditionally been considerable mystery surrounding computers. This mystery has inevitably led to suspicion and fear. Large computer facilities, surrounded by an elite cadre, have not contributed to a better understanding of what computers are and how they work. Small personal computers, such as the Apple II, have made it possible for you to become the master of your own computer. This book will open up the Apple II and show you what makes it tick. You will be able to make it do anything you want. This feeling of control, power, and understanding is something you must experience to fully appreciate. Therefore, let's begin.

2

The 6502 Microprocessor

The 6502 microprocessor is a 40-pin integrated circuit chip (see Figure 2.1) that costs less than $10 and can be used for a wide range of applications. One thing that it can be used for is to build a general-purpose computer like the Apple II. In this book you will learn how this is done and how you can use the 6502, or similar microprocessors, to do many different things. The 6502 microprocessor that is in the Apple II is shown in Figure 2.2. This single chip is the main "brain" of the Apple II. The rest of the chips on the board are mainly memory (see Chapter 3) and other chips related to various input/output (I/O) operations.

THE DATA BUS

A schematic of the 6502 chip is shown in Figure 2.3. This chip is manufactured by MOS Technology, Inc., Rockwell International, and Synertek, Inc.*

---

*MOS Technology, Inc., 950 Rittenhouse Road, Norristown, PA 19401. Rockwell International, Microelectronic Devices, P. O. Box 3669, RC55, Anaheim, CA 92803. Synertek, Inc., 1901 Old Middlefield Way, Mountain View, CA 94043.

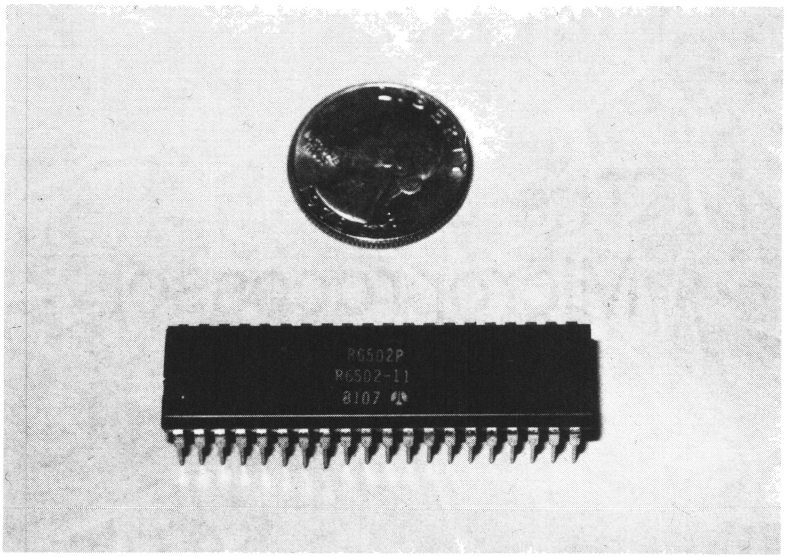


FIGURE 2.1. The 6502 microprocessor.

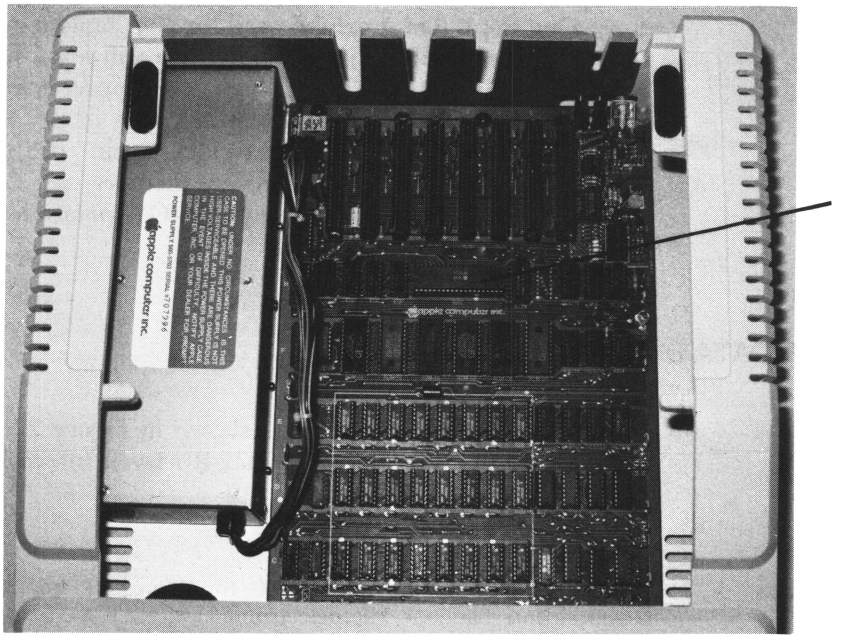


FIGURE 2.2. The 6502 microprocessor is the "brain" of the Apple II.

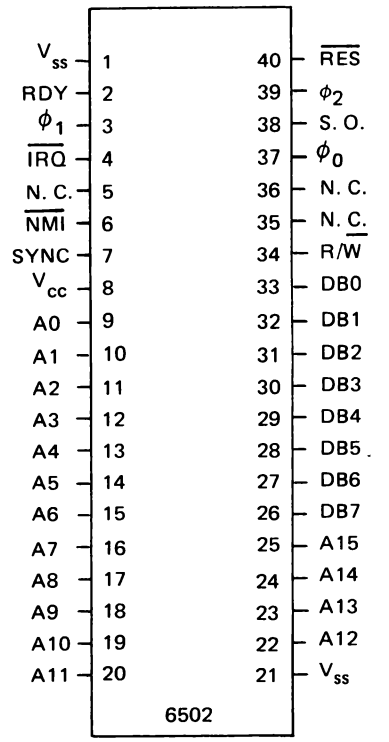
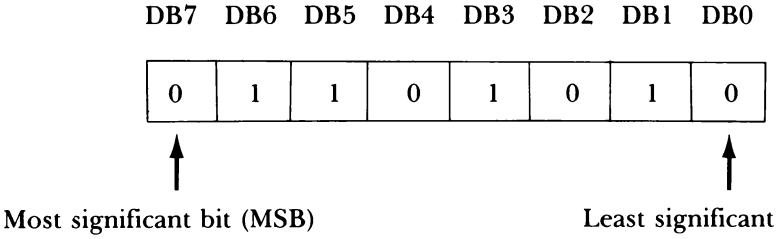


FIGURE 2.3.
Pinout for the 6502 microprocessor.

Pins 26–33, labeled DB0–DB7, form an 8-bit bidirectional *data bus.* These eight lines are connected to various memory chips as described in Chapter 3. All data move between the 6502 microprocessor chip and the memory chips over this data bus in groups of 8 bits called *bytes.* A high voltage (5 volts) on one of these data lines is considered to be a binary digit (bit) 1 and a low voltage (0 volts) is considered to be a binary digit 0. Thus, at some instant of time the data bus might contain the 8 bits, or byte, 01101010. The rightmost bit is DB0, the least significant bit (LSB), and the leftmost bit is DB7, the most significant bit (MSB) (see Figure 2.4).

FIGURE 2.4. A byte (8 bits) of data on the data bus.



When using the 6502 microprocessor, you will always be dealing with 8-bit bytes. It is therefore important to know how to count in binary. It is also convenient to represent binary numbers in a shorthand notation called *hexadecimal.*

COUNTING IN BINARY AND HEXADECIMAL

Consider a box containing one marble. If the marble is in the box, we will say that the box is *full* and associate the digit 1 with the box. If we take the marble out of the box, the box will be empty, and we will then associate the digit 0 with the box. The two binary digits 0 and 1 are called *bits,* and with 1 bit we can count from 0 (box empty) to 1 (box full), as shown in Figure 2.5.

FIGURE 2.5. You can count from 0 to 1 with 1 bit.

Consider now a second box that can also only be full (1) or empty (0). However, when this box is full it will contain *two* marbles as shown in Figure 2.6. With these two boxes (2 bits) we can now count from 0 to 3, as shown in Figure 2.7.

FIGURE 2.6. This box can either contain two marbles (full) or no marbles.

Note that the value of each 2-bit binary number shown in Figure 2.7 is equal to the total number of marbles in the two boxes.

We can add a third bit to the binary number by adding a third box that is full (bit = 1) when it contains four marbles and is empty (bit = 0) when it contains no marbles. It must be either full (bit = 1) or empty (bit = 0). With this third box (3 bits) we can count from 0 to 7, as shown in Figure 2.8.

If you want to count beyond 7 you must add another box. How many marbles should this fourth box contain when it is full (bit = 1)? It should be clear that this box must contain eight marbles. The binary number 8 would then be written as

1000

Remember that a 1 in a binary number means that the corresponding box is full of marbles and the number of marbles that constitutes a full box varies as 1, 2, 4, 8, starting at the right. This means that with 4 bits we can count from 0 to 15, as shown in Figure 2.9.

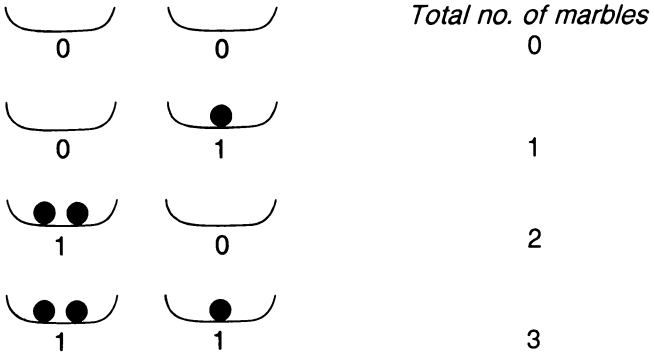


FIGURE 2.7. You can count from 0 to 3 with 2 bits.

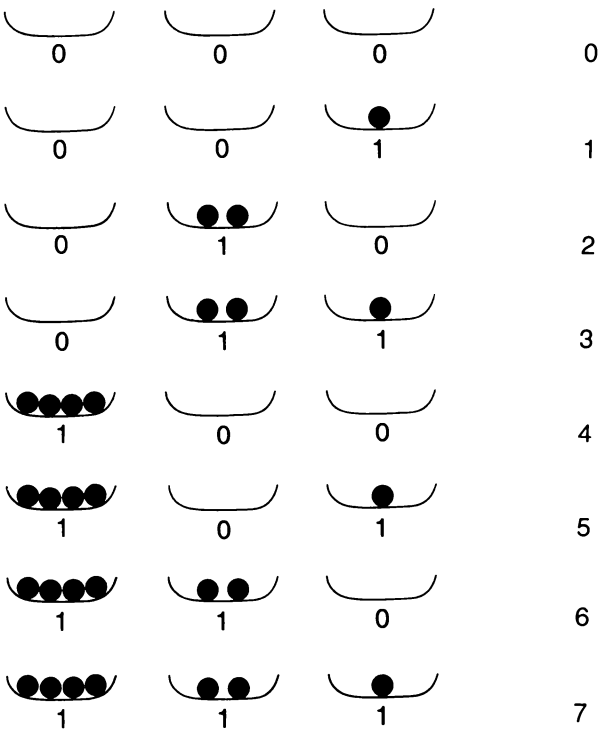


FIGURE 2.8. You can count from 0 to 7 with 3 bits.

It is convenient to represent the total number of marbles in the four boxes represented by the 4-bit binary numbers shown in Figure 2.9 by a single digit. We call this a *hexadecimal* digit; the sixteen hexadecimal digits are shown in the righthand column in Figure 2.9. The hexadecimal digits 0–9 are the same as the decimal digits 0–9. However, the decimal numbers 10–15 are represented by the hexadecimal digits A–F. Thus, for example, the hexadecimal digit D is equivalent to the decimal number 13.

No. of marbles in each full box (bit = 1)				*Total no. of marbles*	*Hex digit*
8	*4*	*2*	*1*		
0	0	0	0	0	0
0	0	0	1	1	1
0	0	1	0	2	2
0	0	1	1	3	3
0	1	0	0	4	4
0	1	0	1	5	5
0	1	1	0	6	6
0	1	1	1	7	7
1	0	0	0	8	8
1	0	0	1	9	9
1	0	1	0	10	A
1	0	1	1	11	B
1	1	0	0	12	C
1	1	0	1	13	D
1	1	1	0	14	E
1	1	1	1	15	F

FIGURE 2.9. You can count from 0 to 15 with 4 bits.

In order to count beyond 15 in binary you must add more boxes. Each full box you add must contain twice as many marbles as the previous full box. With 8 bits you can count from 0 to 255. A few examples are shown in Figure 2.10. The decimal number that corresponds to a given binary number is equal to the total number of marbles in all of the boxes. To find this number, just add up all of the marbles in the full boxes (the ones with binary digits equal to 1).

No. of marbles in each full box (bit = 1)								*Total no. of marbles*
128	*64*	*32*	*16*	*8*	*4*	*2*	*1*	
0	0	1	1	0	1	0	0	52
1	0	1	0	0	0	1	1	163
1	1	1	1	1	1	1	1	255

FIGURE 2.10. You can count from 0 to 255 with 8 bits.

As the length of a binary number increases, it becomes more cumbersome to work with. We then use the corresponding *hexadecimal number* as a shorthand method of representing the binary number. This is very easy to do. You just divide the binary number into groups of 4 bits starting at the right, and then represent each 4-bit group by its corresponding hexadecimal digit given in Figure 2.9. For example, the binary number

$$\underbrace{1001}\underbrace{1010}$$

$$9 \quad A$$

is equivalent to the hexadecimal number $9A. We will often use a dollar sign ($) to indicate a hexadecimal number. You should verify that the total number of marbles represented by this binary number is 154. However, instead of counting the marbles in the "binary boxes" you can count the marbles in "hexadecimal boxes," where the first box contains A × 1 = 10 marbles and the second box contains 9 × 16 = 144 marbles. Therefore, the total number of marbles is equal to 144 + 10 = 154.

A third hexadecimal box would contain a multiple of $16^2 = 256$ marbles and a fourth hexadecimal number would contain a multiple of $16^3 = 4{,}096$ marbles. As an example, the 16-bit binary number

$$\underbrace{1000}\underbrace{0111}\underbrace{1100}\underbrace{1001}$$

$$8 \quad 7 \quad C \quad 9$$

is equivalent to the decimal number 34,761 (that is, it represents 34,761 marbles). This can be seen by expanding the hexadecimal number:

$$\begin{aligned} 8 \times 16^3 &= 8 \times 4{,}096 = 32{,}768 \\ 7 \times 16^2 &= 7 \times 256 = 1{,}792 \\ C \times 16^1 &= 12 \times 16 = 192 \\ 9 \times 16^0 &= 9 \times 1 = 9 \\ & \qquad\qquad\qquad\quad 34{,}761 \end{aligned}$$

You can see that by working with hexadecimal numbers you can reduce by a factor of 4 the number of digits that you have to work with.

Table 2.1 will allow you to conveniently convert hexadecimal numbers of up to four digits to their decimal equivalent. Note, for example, how the four terms in the conversion of $87C9 can be read directly from the table. Each hex digit is read from one column in the table starting at the left. Recall that a byte contains 8 bits, or two hexadecimal characters. Table 2.1 can therefore be used to convert hexadecimal numbers containing up to 2 bytes, or four hexadecimal characters, or 16 bits.

Number Systems

Binary numbers are numbers to the base 2 and hexadecimal numbers are numbers to the base 16. A number N can be written in any base b using the following positional notation:

$$N = p_4p_3p_2p_1p_{0_b} = p_4b^4 + p_3b^3 + p_2b^2 + p_1b^1 + p_0b^0$$

where the number always starts with the least significant digit on the right.

For example, the decimal number 584 is a base 10 number, and can be expressed as

$$\begin{aligned} 584_{10} &= 5 \times 10^2 + 8 \times 10^1 + 4 \times 10^0 \\ &= 500 + 80 + 4 \\ &= 584_{10} \end{aligned}$$

A number to the base b must have b different digits. Thus, decimal numbers (base 10) use the 10 digits 0–9.

A binary number is a base 2 number and therefore uses only the two digits 0 and 1. For example, the binary number 110100 is the base 2 number:

$$\begin{aligned} 110100_2 &= 1 \times 2^5 + 1 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ &= 32 + 16 + 0 + 4 + 0 + 0 \\ &= 52_{10} \end{aligned}$$

This is the same as the first example in Figure 2.10 where the total number of marbles is 52 (32 + 16 + 4).

A hexadecimal number is a base 16 number and therefore needs 16 different digits to represent the number. We use the ten digits 0–9 plus the six letters A–F as shown in Figure 2.9. For example, the hexademical number 3AF can be written as the base 16 number:

$$\begin{aligned} 3AF_{16} &= 3 \times 16^2 + A \times 16^1 + F \times 16^0 \\ &= 3 \times 256 + 10 \times 16 + 15 \times 1 \\ &= 768 + 160 + 15 \\ &= 943_{10} \end{aligned}$$

Microcomputers move data around in groups of 8-bit binary bytes. Therefore, it is natural to describe the data in the computer as binary, or base 2, numbers. As we have seen, this is simplified by using hexadecimal numbers where each hex digit represents four binary digits. Some larger computers represent binary numbers in groups of three bits rather than four. This resulting number is an octal, or base 8, number. Octal numbers use only the 8 digits 0–7. For example, the octal number 457 can be written as the base 8 number

$$\begin{aligned} 457_8 &= 4 \times 8^2 + 5 \times 8^1 + 7 \times 8^0 \\ &= 256 + 40 + 7 \\ &= 303_{10} \end{aligned}$$

We will use only binary and hexadecimal numbers to describe the data in the Apple. The examples in this section show how you can convert a number in any base to a decimal number. Later in this chapter we will look at how to convert a decimal number to hexadecimal.

Table 2.1 Hexadecimal and Decimal Conversion

15 BYTE 8				7 BYTE 0			
15 CHAR 12		11 CHAR 8		7 CHAR 4		3 CHAR 0	
HEX	DEC	HEX	DEC	HEX	DEC	HEX	DEC
0	0	0	0	0	0	0	0
1	4,096	1	256	1	16	1	1
2	8,192	2	512	2	32	2	2
3	12,288	3	768	3	48	3	3
4	16,384	4	1,024	4	64	4	4
5	20,480	5	1,280	5	80	5	5
6	24,576	6	1,536	6	96	6	6
7	28,672	7	1,792	7	112	7	7
8	32,768	8	2,048	8	128	8	8
9	36,864	9	2,304	9	144	9	9
A	40,960	A	2,560	A	160	A	10
B	45,056	B	2,816	B	176	B	11
C	49,152	C	3,072	C	192	C	12
D	53,248	D	3,328	D	208	D	13
E	57,344	E	3,584	E	224	E	14
F	61,440	F	3,840	F	240	F	15

THE INTERNAL REGISTERS

The 6502 microprocessor has several internal registers that can store data. These are shown in Figure 2.11. The accumulator A, the two index registers X and Y, the status or condition code register C, and the stack pointer SP are all 8-bit registers. The program counter PC is a 16-bit register. A more detailed discussion of these registers will be given in Chapter 4.

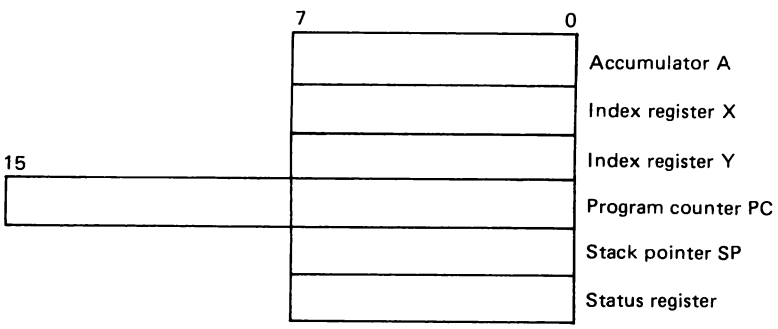


FIGURE 2.11. The internal registers in the 6502 microprocessor.

If you run the monitor program you will obtain the screen display shown in Figure 2.12. The current contents of all of the internal registers are shown on the top half of the screen. The contents of the accumulator A and the X and Y index registers are displayed in both binary and hexadecimal.

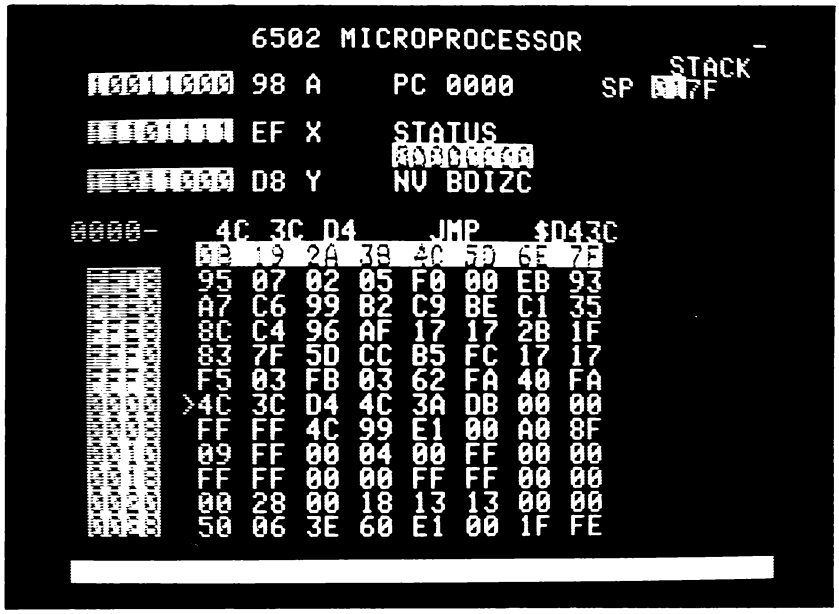


FIGURE 2.12. Screen display of monitor program.

The white horizontal line across the bottom of the screen is called the *command line.* When you press the key containing the slash (/), the message shown at the top of page 13 will appear on the command line, as shown in Figure 2.13:

FIGURE 2.13. Press the slash key, /, to initiate a command.

COMMAND: BDEFILMOPRSTQZ

The computer is now waiting for you to press a key containing one of the letters following the word COMMAND. You will learn what all of these letters do as we progress through the book. For now press the key R. This will produce the message

REGISTER CHANGE: A X Y P C S

on the command line, as shown in Figure 2.14. You can now change the contents of any of the six registers by pressing the appropriate key.

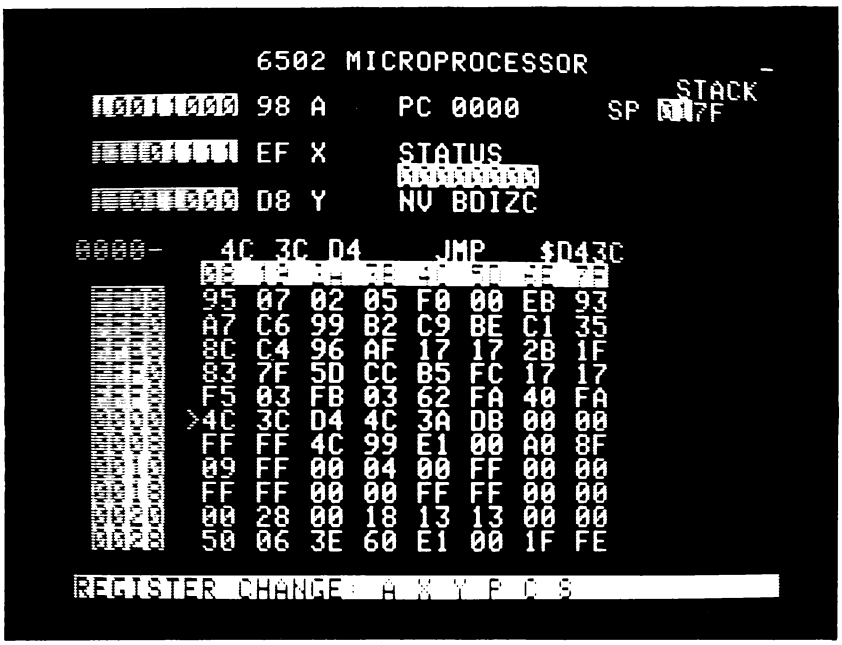


FIGURE 2.14. /R is used to change the contents of the internal registers.

For example, to change the contents of the accumulator, press key A. This will produce the entry A= followed by a blinking cursor on the bottom line of the screen (this is called the entry line). Type in any two-digit hexadecimal number. When you press the RETURN key this new number will be stored in accumulator A in both hexadecimal and binary format. Try it. If you enter more, or fewer, than two hexadecimal digits, or if you enter a digit other than 0–9 or A–F, the computer will ignore your request, give a beep, and clear the command and entry lines. You can then type /R again to change a register value.

If you make a mistake while entering a hexadecimal value on the entry line, you can back the cursor up (and thereby erase the most recently entered value) by pressing the ESC key. You can't use the backspace key (←) for this purpose because we use this key to move the memory cursor around in memory (see Chapter 3).

To change a value in the index register X, type /RX. Then enter a two-digit hexadecimal value as you did for the accumulator A. Try changing the contents of X to 00.

To change the value in the index register Y, type /RY, and enter a two-digit hexadecimal value. Try changing the contents of Y to FF. Note that this stores eight 1s in the Y index register.

To change the value in the status, or condition code register CC, type /RC. Then enter a two-digit hexadecimal value. Note that this hex value will be displayed on the screen as the corresponding binary value.

EXERCISE 2.1

What two-digit hexadecimal number must you store in accumulator A in order for the following 8 bits to be stored in A? Verify your answers by changing the contents of A on the screen to each of the following values:

1. 01110010
2. 00011111
3. 11001101
4. 00111001
5. 11101000
6. 10110100
7. 10100101
8. 00000110

EXERCISE 2.2

What 8-bit binary number is equivalent to the following hexadecimal numbers? Verify your answers by changing the contents of the index register X to each of these values and noting its binary value on the screen.

1. $65
2. $0A
3. $48
4. $BE
5. $9D
6. $C3
7. $F2
8. $17

DECIMAL–HEXADECIMAL CONVERSION

Suppose that you wish to store the decimal value 167_{10} (base 10) in accumulator A. What hexadecimal value (base 16) should you enter? The easiest way to figure this out is to look at Table 2.1. The closest decimal value in this table that does not exceed 167 is 160, in the second column from the right. This corresponds to the hexadecimal digit A as the second digit from the right ($A \times 16^1 = 10 \times 16 = 160$). To find the hexadecimal digit to use in the rightmost position, just subtract 160 from 167. Thus the decimal number 167_{10} is equivalent to the hexadecimal number $A7_{16}$. What binary number is this?

How can you convert a decimal number to hexadecimal if you don't have Table 2.1 around? Here's a shortcut. Divide the decimal number by 16 and keep track of the remainder. Keep dividing the results by 16 and writing down the remainders at each step until the result is 0. The equivalent hexadecimal number is just all of the remainders read backward. For example, this is how to convert the decimal number 167_{10} to hexadecimal:

```
167/16 =    10    with remainder 7
 10/16 =     0    with remainder 10 = A
                  read backward ____^
167(10) =  A7(16)
```

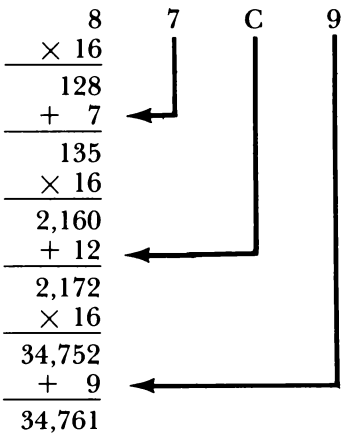
Here's the example given just before Table 2.1 earlier in this chapter:

```
34,761(10) = ?(16)

34,761/16 = 2,172    with remainder 9
 2,172/16 =   135    with remainder 12 = C
   135/16 =     8    with remainder 7   ^
     8/16 =     0    with remainder 8   |
                     read up ___________|
```

Therefore, $34,761_{10} = 87C9_{16}$

There is a similar shortcut for going the other way. To convert the hexadecimal number $A7_{16}$ to decimal, multiply A $\times$ 16 and add 7. For longer hexadecimal numbers, start with the leftmost digit (the most significant), multiply it by 16, and add the next hex digit. Multiply this result by 16 and add the next hex digit. Continue this process until you have added the rightmost digit. For example, to convert $87C9_{16}$ to decimal, you can do this:



Therefore, $87C9_{16} = 34{,}761_{10}$

EXERCISE 2.3

Convert the following decimal numbers to their hexadecimal equivalent:

1. 42
2. 125
3. 249
4. 2,173
5. 31,729
6. 62,433

EXERCISE 2.4

Convert the following hexadecimal numbers to their decimal equivalent:

1. $EF
2. $5C
3. $AA
4. $5AC
5. $7134
6. $F21C

3

Computer Memory

The 6502 microprocessor chip by itself is not very useful. It must be connected to some external memory chips that can store both data and instructions (the program) for the 6502. In this chapter you will learn

1. how the 6502 microprocessor sends and receives bytes of data from different memory locations
2. how to use the TUTOR monitor to examine and change the contents of any memory location
3. how to enter both hex and ASCII data into memory

THE ADDRESS BUS

Pins 9–20 and 22–25 of the 6502 chip labeled A0–A15 in Figure 2.3 form a 16-bit *address bus*. This bus produces a 16-bit address which addresses one of $2^{16} = 65{,}536$ possible memory locations. This number is equal to 64K, where 1K (from kilo) = 1,024. Each memory location contains one 8-bit byte of data. These data move between the 6502 chip and a memory chip on the data bus DB0–DB7.

Memory chips come in various sizes and configurations. For example, the Motorola 6810 static RAM chip contains 128 bytes of memory. Seven address lines are needed to address all 128 bytes ($2^7 = 128$). The low-order seven-address lines A0–A6 from the 6502 chip are connected to the address pins on the memory chip as shown in Figure 3.1. If the low-

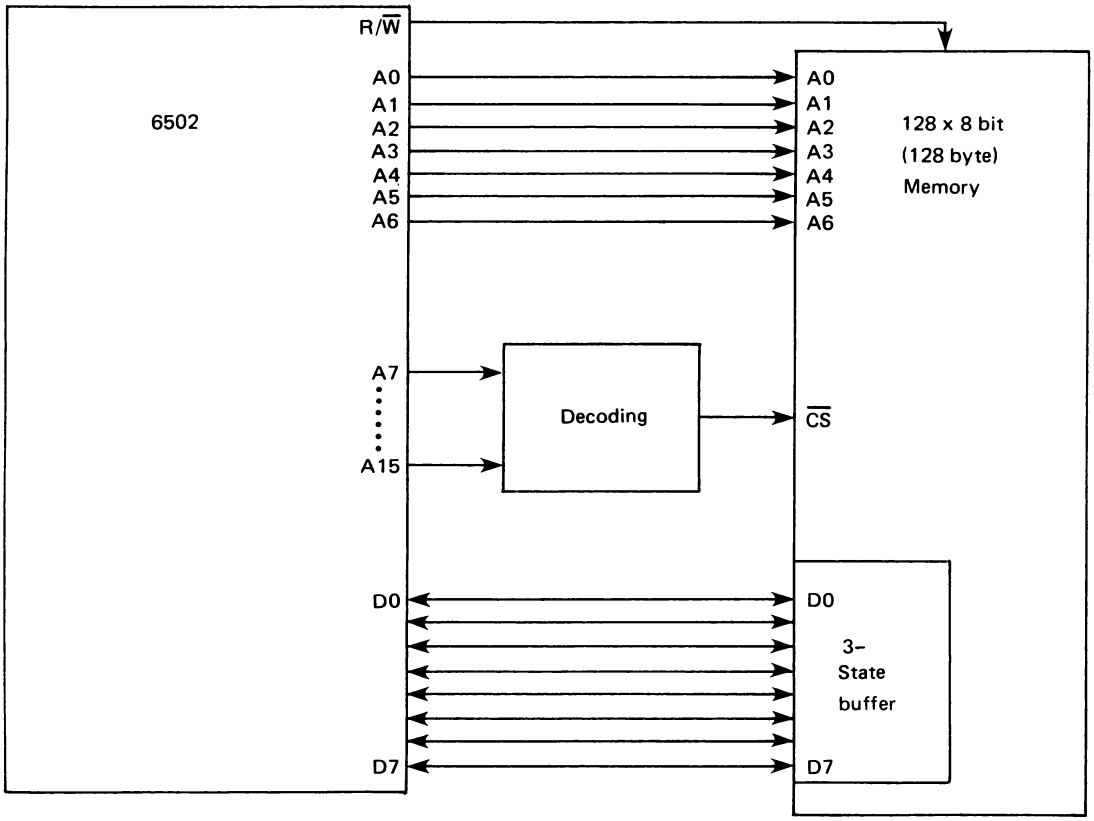


FIGURE 3.1. Connecting the 6502 microprocessor chip to a memory chip.

order lines A0–A6 are used, the memory chip will contain 128 consecutive addresses.

Exactly which addresses the memory chip will respond to depends on how the high-order address lines A7–A15 are decoded. Each memory chip has at least one chip select pin $\overline{CS}$. When this pin is low (0 volts) the data lines D0–D7 are connected to the internal memory of the memory chip; therefore, the 6502 microprocessors can *read* the contents of a memory location when the read/write line R/$\overline{W}$ is high. When the $\overline{CS}$ pin is high, an internal *three-state buffer* connected to the data bus is put into a high-impedance state.* This effectively produces an open circuit between the data lines D0–D7 and the internal memory in the chip. This means that the 6502 will not "see" this memory chip.

By having the data lines connected to the internal memory through three-state buffers, more than one memory chip can be connected to the

* A *three-state buffer* allows data to be in one of three states: high (1), low (0), or high-impedance.

same data bus. As long as only one chip has its $\overline{CS}$ pin low at any one time, only one chip (and therefore one memory location) will be addressed for a particular address on the address lines A0–A15. It is up to the decoding of the high-order address lines to ensure that only one memory chip is selected for a particular address. We will look at how this is done in Chapter 13.

The 16-bit address put out by the 6502 microprocessor can be represented by a four-digit hexadecimal number between $0000 and $FFFF. The TUTOR monitor allows you to see what data byte is stored in any of these 64K memory locations.

THE TUTOR'S MEMORY DISPLAY

When you run the TUTOR monitor program, the screen display is as shown in Figure 3.2. Along the left side of the screen are eleven rows of reverse video four-digit hexadecimal addresses. Note that the addresses of adjacent rows differ by 8. The sixth row contains the address $0000. The 8 bytes following this address (remember that each byte is represented by a two-digit hexadecimal number) are the contents of memory locations $0000–$0007. The 8 bytes in the next row (following the address $0008) are the contents of memory locations $0008–$000F.

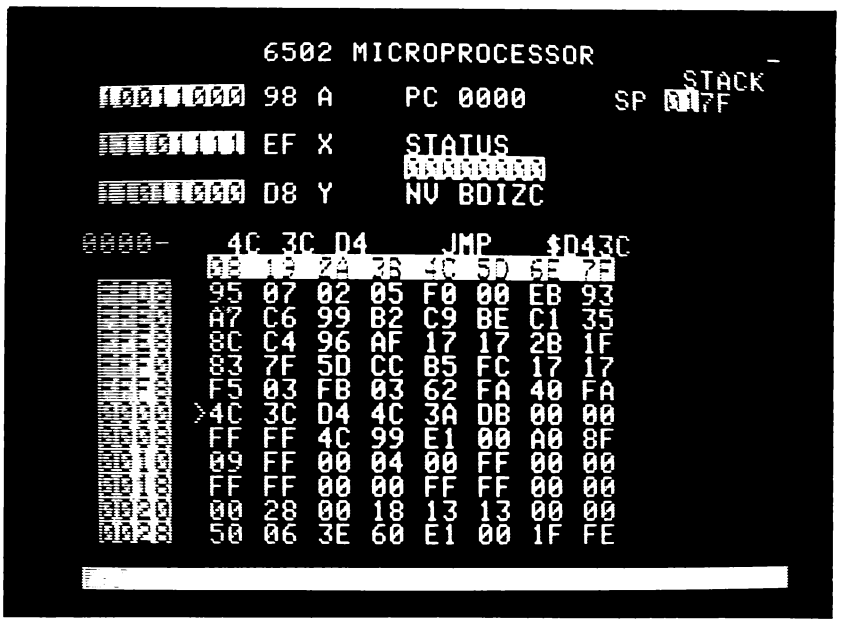


FIGURE 3.2. The TUTOR displays the contents of 88 consecutive memory locations.

The white bar at the top of the memory display contains the digits

08	19	2A	3B	4C	5D	6E	7F

This is a guide that gives the last digit of a memory address. Note that the starting address on each line ends with either a 0 or an 8. If it ends with a 0, then the addresses of the 8 bytes on that line end with the *first* digit in each pair (0–7) of the guide line shown here. If the starting address ends with an 8, then the addresses of the 8 bytes on that line end with the *second* digit in each pair (8–F) of the guide line.

A position cursor, >, is pointing to the contents, 4C, of memory location $0000. Note that the preceding line begins with address $FFF8 and ends with $FFFF. Thus, this display shows the contents of the bottom of memory from locations $0000 to $002F and the very top of memory from locations $FFD8 to $FFFF. It is possible to look at the contents of any memory location in between by scrolling the display.

Scrolling through Memory

Look at the upper-right-hand corner of the screen. You should see a dash (—). Press the space bar. The dash should change to an exclamation point (!). Press the space bar again. It should return to a dash.

Now press the key containing the right arrow → and watch the position cursor >. Note that it moves to memory location $0001. Press the → key several more times. Note that when you get to the end of the line (location $0007) and advance to location $0008, all of the memory locations scroll up one line, and a new line containing addresses $0030–$0037 appears at the bottom of the display.

Press the key containing the backward arrow ←. Note that you back up 1 byte in memory each time you press the ← key.

Now press the space bar so that the exclamation point (!) appears in the upper-right-hand corner of the screen. If you now press the → key, all memory addresses will scroll up one line. That is, the position cursor will advance by 8 bytes. If you press the → key while holding down the REPT key you will continuously scroll forward in memory. Try it.

Now press the ← key. This will cause all memory addresses to scroll down one line. That is, the position cursor will back up 8 bytes. If you press the ← key while holding down the REPT key, you will continuously scroll backward in memory. Try it.

By pressing the space bar you can go back and forth, advancing or backing up 1 bytre (—) or one row (!) when you press the forward → or backward ← keys. Play with these keys until you become familiar with how they work.

Go To (>) Any Memory Location

Scrolling through memory is not a convenient way to get to a memory location that is a long way off. You can use the > symbol to go to any memory location you want. Type > by pressing the period key while holding down the SHIFT key. The message

GOTO: MEMORY ADDRESS

will appear on the command line and a blinking cursor will appear on the entry line, as shown in Figure 3.3.

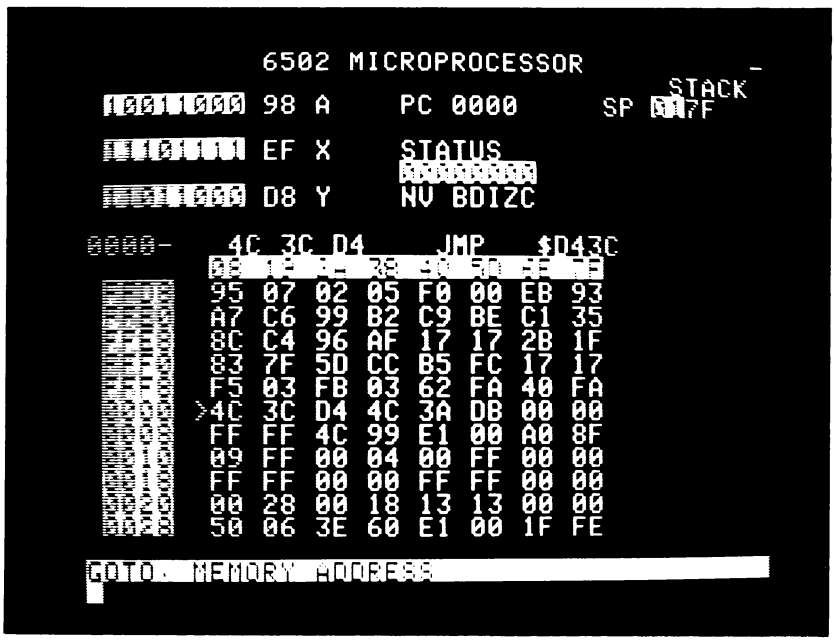


FIGURE 3.3. Press > to go to any memory location.

Enter any address and press RETURN. The position cursor will immediately move to the memory address that you entered. You do not have to type leading 0s when entering the address. For example, if you enter B, the position cursor will move to address $000B.

If you make a mistake while entering an address, pressing the ESC key will move the blinking cursor back one space and erase the most recently entered digit. Try this. If you back up the blinking cursor beyond the starting position, a beep will sound, the command line will clear, and you will return to the TUTOR monitor.

If you type in an invalid address (more than four hex digits or any digits other than 0–F), a beep will sound when you press RETURN, and you will return to the TUTOR monitor. Try this.

EXERCISE 3.1

Examine the contents of memory locations 68, 40A, 800E, FDC4, and 21.

Changing the Contents of Memory

The TUTOR makes it easy for you to change the contents of memory locations. The 6502 programs you will write will consist of a sequence of bytes that you must store in consecutive memory locations. Therefore, it will be convenient to be able to easily change the contents of consecutive memory locations. This can be done by using the command /M.

Go to memory location $800 by typing >800 RETURN. Then type /M. This will produce the following message on the command line (see Figure 3.4a):

MEMORY CHANGE: H A

You have two choices at this point. You can enter hex data by pressing key H, or you can enter ASCII data by pressing key A. We will consider ASCII data in the next section. For now, press key H. The message ENTER HEX VALUES will appear on the command line (see Figure 3.4b). Type

11 22 33 44

as shown in Figure 3.4b, and note that these hex values are stored in memory locations $800–$803. Each time you type a byte it appears on the entry line and is stored in the memory location pointed to by the position cursor. The position cursor is then automatically advanced to the next memory location. You can enter as many hex bytes as you want. When you come to the end of an entry line, it will automatically clear and start again at the beginning of the line. Try this. When you finish entering hex values, just press RETURN. Pressing any nonhex key while entering hex values will produce a beep and a return to the TUTOR monitor.

EXERCISE 3.2

Store the following data in memory starting at location $800:

AA A8 CA 88 88 8A 98 E8 E8 C8 98 8A

ASCII Data

The name ASCII stands for "American Standard Code for Information Interchange." In this standard code a certain 7-bit binary number is associated with each character (letter, digit, or special character). This code is

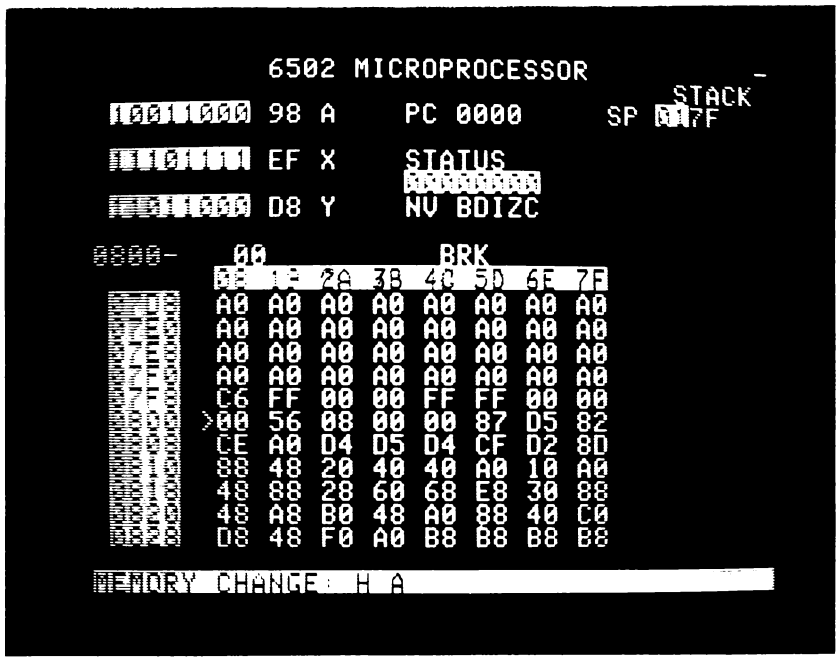


FIGURE 3.4. (a) Press /M to change memory values.

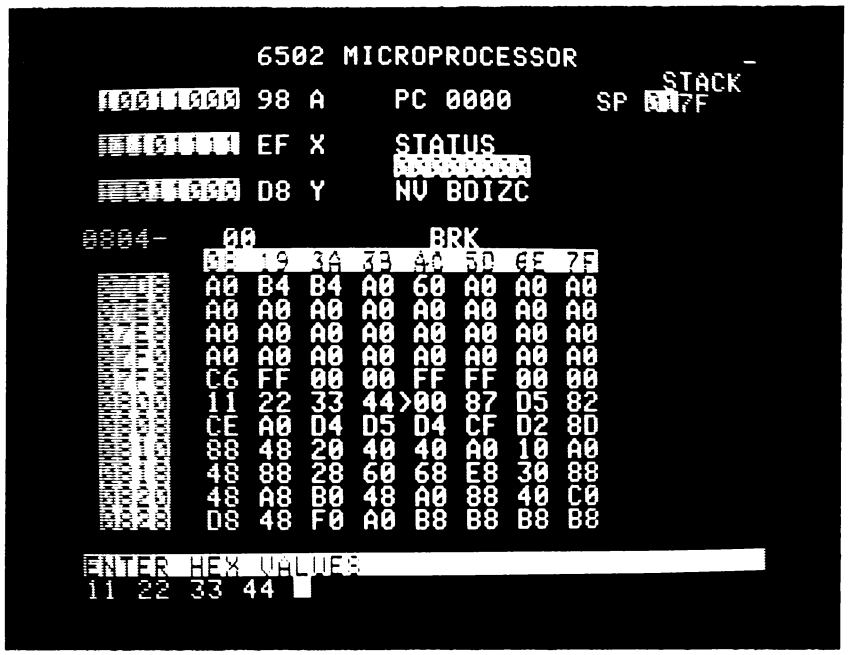


FIGURE 3.4. (b) Press /MH to enter hex values into memory.

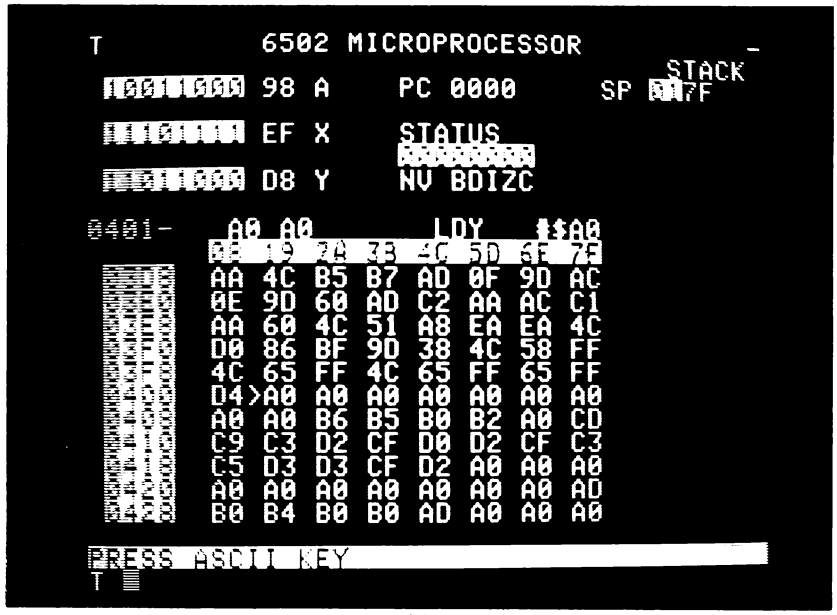
used extensively throughout the industry for sending information from one computer to another or for sending data from a terminal to a computer. The hex values for the ASCII codes of all characters are shown in Figure 3.5. To use this chart, read the high-order nibble (bits 4–7) across the top and the low-order nibble (bits 0–3) along the left side of the chart. For example, the ASCII code for A is $41.

HEX	*MSD*	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*
LSD	*BITS*	*000*	*001*	*010*	*011*	*100*	*101*	*110*	*111*
0	0000	NUL	DLE	SPACE	0	@	P		p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	--
D	1101	CR	GS	-	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	←	o	DEL

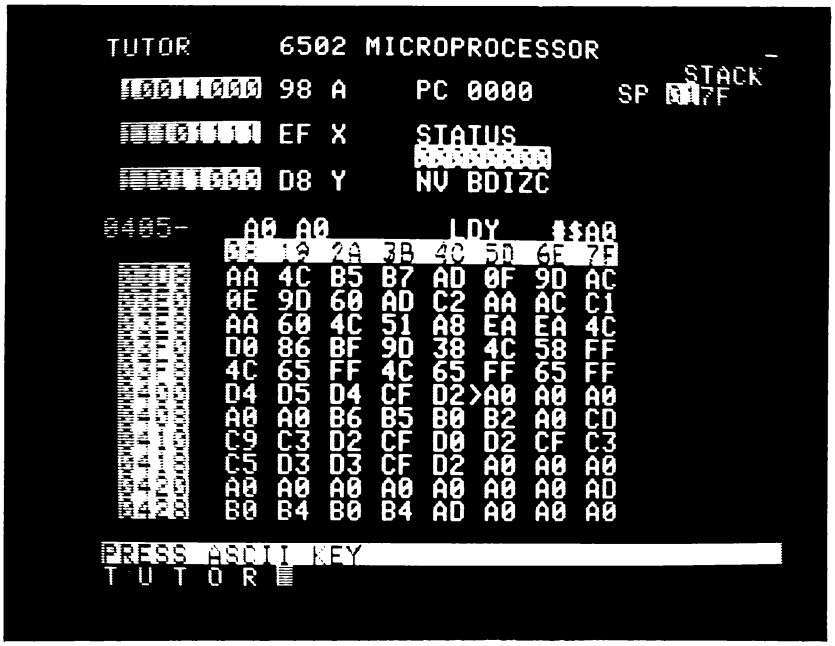
FIGURE 3.5. Standard ASCII codes.

Decimal:		128	144	160	176	192	208	224	240
Hex:		$80	$90	$A0	$B0	$C0	$D0	$E0	$F0
0	$0	nul	dle		0	@	P		p
1	$1	soh	dc1	!	1	A	Q	a	q
2	$2	stx	dc2	"	2	B	R	b	r
3	$3	etx	dc3	#	3	C	S	c	s
4	$4	eot	dc4	$	4	D	T	d	t
5	$5	enq	nak	%	5	E	U	e	u
6	$6	ack	syn	&	6	F	V	f	v
7	$7	bel	etb	'	7	G	W	g	w
8	$8	bs	can	(	8	H	X	h	x
9	$9	ht	em	)	9	I	Y	i	y
10	$A	lf	sub	*	:	J	Z	j	z
11	$B	vt	esc	+	;	K	[	k	{
12	$C	ff	fs	,	<	L	\	l	|
13	$D	cr	gs	-	=	M	]	m	}
14	$E	so	rs	.	>	N	^	n	~
15	$F	si	us	/	?	O	_	o	rub

FIGURE 3.6. Apple II ASCII codes.



(a)



(b)

FIGURE 3.7. Press /MA to enter ASCII values into memory.

Note that bit 7 is assumed to be 0 in Figure 3.5. This is because standard ASCII code uses only 7 bits (0–6). The eighth bit (bit 7) is often used as an error-checking parity bit when sending data from a terminal to a computer.

The Apple II computer, however, uses this eighth bit as part of the ASCII code for each character. This will allow more characters to be defined by an ASCII code. In the Apple II computer, reverse video and flashing characters have their own unique ASCII code. The ASCII codes for normal characters used by the Apple II are shown in Figure 3.6. By comparing Figures 3.5 and 3.6 you can see that the Apple II ASCII codes differ from the corresponding standard ASCII codes by having bit 7 set to 1 rather than 0.

The TUTOR makes it easy for you to enter Apple II ASCII codes into particular memory locations. As an example, go to memory location $400 by typing >400. Memory location $400 is the first memory location of the TV RAM. Each memory location in the TV RAM corresponds to a particular location on the TV screen and contains the ASCII code for the character that is currently being displayed at that location on the screen.

Memory location $400 corresponds to the upper-left-hand corner of the TV screen. Note that it contains the hex value A0. From Figure 3.6 you see that this is the ASCII code for a blank, which is what is being displayed at the upper-left-hand corner of the screen. You can change this value by typing /MA. The message PRESS ASCII KEY is displayed on the command line. Press key T. Note that the ASCII code for T, $D4, is stored in memory location $400, which causes the T to be displayed in the upper-left-hand corner of the screen, as shown in Figure 3.7a.

The T is also displayed on the entry line (the bottom line of the screen) and the position cursor advances to memory location $401. You can continue the enter ASCII values. For example, type TUTOR as shown in Figure 3.7b. Note that the word TUTOR appears on the top line of the screen as the ASCII codes for each letter are stored in memory locations $400–$404. We will take a closer look at how characters are displayed on the screen in Chapter 9.

4

The 6502 Registers

The TUTOR displays the contents of the 6502 registers on the top half of the screen. In Chapter 2 you saw how to change the contents of any register by typing /R followed by the register whose contents you want to change. In this chapter we will take a closer look at each register. In particular, you will learn

1. how to execute a single instruction with the TUTOR
2. how the instructions ASL, LSR, ROL, and ROR affect accumulator A
3. how the instructions TAX, TXA, DEX, and INX affect the index register X
4. how the instructions TAY, TYA, DEY, and INY affect the index register Y
5. The meanings of the bits in the status register and the instructions CLC, SEC, CLD, SED, CLI, SEI, and CLV
6. how the instructions TSX and TXS affect the stack pointer

ACCUMULATOR A

The accumulator A is a general-purpose 8-bit register that is used primarily for storing intermediate results. When you want to move a byte from one memory location to another you must first load the byte into one of the registers such as accumulator A (with a load accumulator A, LDA, in-

struction) and then store the byte in the other memory location (with a store accumulator A, STA, instruction). The details of how to address various memory locations will be described in Chapter 8.

Addition and subtraction using instructions ADC (add with carry) and SBC (subtract with carry) always involve addition to or subtraction from a value stored in accumulator A. The result is always stored in accumulator A. The details of adding and subtracting numbers will be given in Chapter 5.

Each 6502 instruction has an 8-bit operation code or *op-code*, associated with it. For example, if you want a program to load the hex value $7A into accumulator A, you would have to store the 2 bytes $A9 $7A in memory. The hex value $A9 is the op-code for LDA (load accumulator A) immediate mode. This means that accumulator A must be loaded with the hex value ($7A) stored in the next memory location. When the op-code A9 is executed, the following value, 7A, will then be stored in accumulator A.

To see this, go to memory location $300 by typing >300 and then enter A9 7A in locations $300 and $301 by typing /MH A9 7A.

Now move the position cursor (>) back to memory location $300, which contains the op-code A9. If you type CTRL S (press key S while holding down the CTRL key), the Apple II will execute the *single instruction* at the position cursor (A9). Try it. The hex value 7A should have been stored in accumulator A, as shown in Figure 4.1. Note that the po-

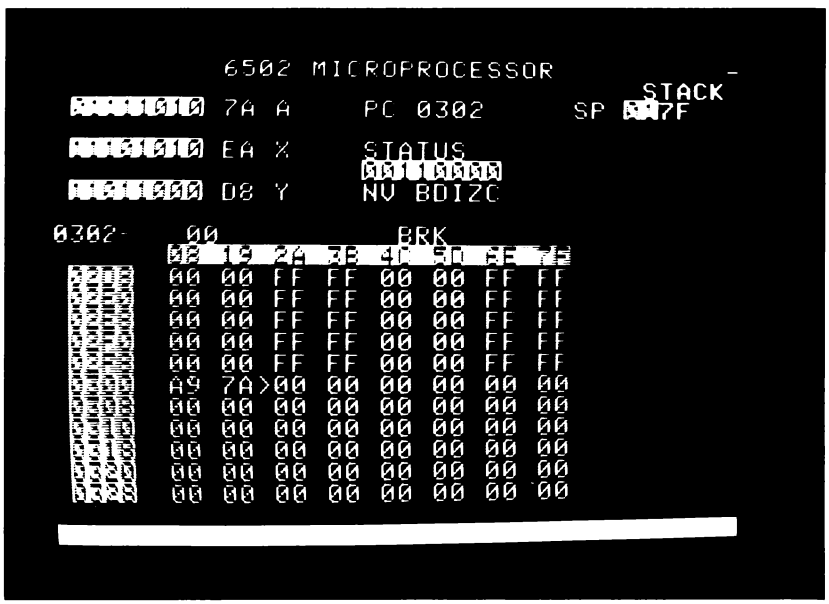


FIGURE 4.1. Pressing CTRL S will execute a single instruction at the cursor location.

sition cursor moves to location $302, which would normally contain the op-code for the next instruction to be executed.

We will look at other instructions related to accumulator A later in this chapter. First let's take a closer look at the program counter.

PROGRAM COUNTER (PC)

The program counter, PC, is a 16-bit register that contains the address of the next instruction to be executed. When an instruction is executed, the progam counter is automatically incremented the number of times needed to point to the next instruction. Instructions may be 1, 2, or 3 bytes long. Therefore, the program counter may be incremented by 1, 2, or 3 depending upon the instruction being executed.

You can change the value stored in the program counter by pressing /RP and then entering a new four-digit hex value. Normally, however, the program counter will automatically change itself as a program is executed.

For example, the program shown in Figure 4.2 will successively load the hex values $12, $34, and $56 into accumulator A and then execute two NOPs (no operation). The NOP instruction will only advance the program counter. Note that in Figure 4.2 the machine code instruction A9 12 can be written in *assembly language* as LDA #$12. The # sign stands for "immediate mode"; that is, LDA #$12 means "Load accumulator A with the hex value $12." Assembly language instructions are easier for a person to understand than the corresponding machine code. However, the 6502 microprocessor only understands the machine code. Therefore, once you write an assembly language program you (or the computer) must convert it into machine code before it can be executed. A program that does this conversion for you is called an *assembler.* In this book you will do the conversion yourself by looking up the op-codes for the various instructions, using the tables given in Appendix A. You can then enter the machine code and execute the program using the TUTOR monitor.

Memory Address	*Machine Code Instructions*	*Assembly Language Instructions*
0300	A9 12	LDA #$12
0302	A9 34	LDA #$34
0304	A9 56	LDA #$56
0306	EA	NOP
0307	EA	NOP

FIGURE 4.2. Single stepping through this program will cause the program counter to advance to the next instruction.

For example, enter the program shown in Figure 4.2 by going to memory location $300 and then typing

/MH A9 12 A9 34 A9 56 EA EA (RETURN)

Then move the position cursor back to location $300 and press CTRL S five times. Note that each time you press CTRL S a single instruction is executed and the program counter advances to the next instruction. Also note that the assembly language version of the next instruction to be executed is displayed above the memory portion of the screen (see Figure 4.3). Note also that the NOP instruction (op-code EA) does nothing except advance the program counter by 1. The NOP instruction takes about 2 μsec (2×10^{-6} seconds) to execute; it is sometimes used when short delays of a few microseconds are required.

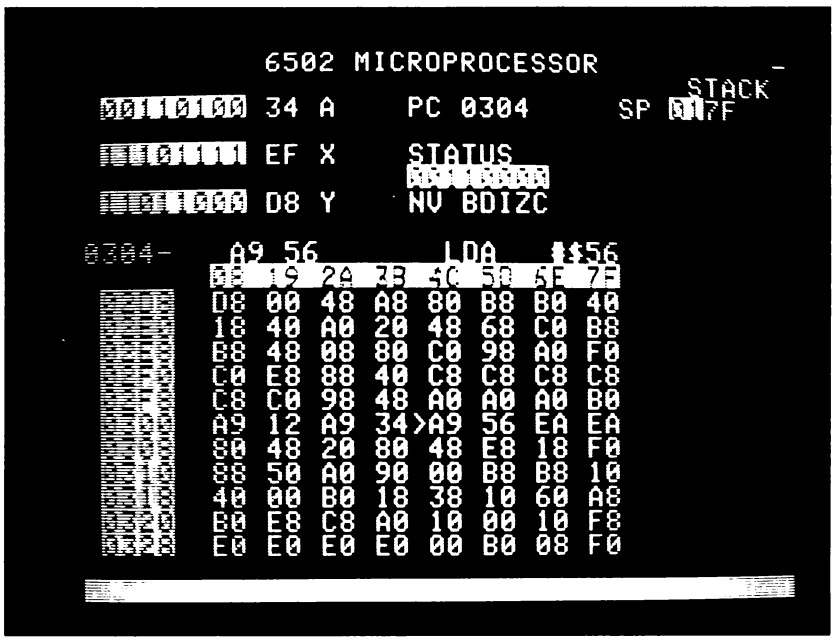


FIGURE 4.3. The assembly language version of the instruction at the position cursor is displayed above the memory portion of screen.

STATUS OR CONDITION CODE REGISTER (CC)

The 6502 has a status register (we will also refer to it as a condition code register) that contains four status flags and three condition flags. The four status flags are the carry flag (C), the zero flag (Z), the negative flag (N), and the overflow flag (V). The three condition flags are the interrupt dis-

able flag (I), the break flag (B), and the decimal mode flag (D). Each flag is 1 bit in the status register. The location of each flag is shown in Figure 4.4 and the value of each flag is displayed in the status register shown near the top center of the TUTOR screen.

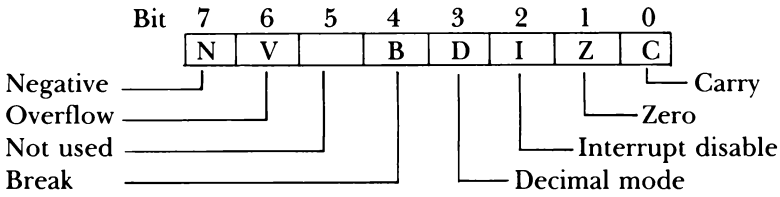


FIGURE 4.4. The status or condition code register.

You can change the contents of the condition code register by typing /RC and then entering a hexadecimal value. For example, if you enter 93, the bit pattern 10010011 will be displayed in the status register. Try it.

We will now look at the meaning of each flag.

Carry (C)

The carry flag is bit 0 of the status register. It can be considered to be an extension of accumulator A, the index registers X or Y, or a memory location operated on by an instruction. The carry bit is changed by three different types of instructions. The first are instructions involving addition and subtraction. These include ADC (add with carry) and SBC (subtract with carry) described in Chapter 5 and CMP (compare accumulator A), CPX (compare index register X), and CPY (compare index register Y) described in Chapter 6.

The second group of instructions that can change the carry bit are the shifting and rotating instructions (ASL, LSR, ROL, ROR) that will be described later in this chapter.

Finally the carry bit is set to 1 with the instruction SEC (set carry; op-code = 38) and is cleared to 0 with the instruction CLC (clear carry; op-code = 18).

EXERCISE 4.1

The op-codes for CLC and SEC are 18 and 38, respectively. Enter the following bytes in memory, starting at location $300:

18 38 18 38 18 38

Single step through these instructions (by pressing CTRL S) and watch the carry flag.

Zero Flag (Z)

The zero flag is bit 1 of the status register. This flag is set to 1 when the result of an instruction is 0. If the result of an instruction is not 0, the Z flag is cleared to 0. This Z flag is tested by the branching instruction BEQ (branch if equal to 0, Z = 1) and BNE (branch if not equal to 0, Z = 0). We will describe how these branching instructions work in Chapter 6.

Negative Flag (N)

The negative flag is bit 7 of the status register. Negative numbers are stored in 6502 computers using a two's complement representation. In this representation a negative number is indicated when bit 7 (the leftmost bit) of a byte is set to 1. We will look at the two's complement representation of a number in detail in Chapter 5. When the result of an instruction leaves bit 7 (the sign bit) set, the N flag is set to 1. If the result of an instruction is positive (bit 7 = 0), the N flag is cleared to 0. The N flag is tested by the branching instructions BMI (branch on minus, N = 1) and BPL (branch on plus, N = 0). We will discuss these branching instructions in Chapter 6.

Overflow Flag (V)

The overflow flag is bit 6 of the status register. It is set any time the 8-bit result of a signed (two's complement) operation is outside the range − 128 through +127. Its relationship to the carry flag will be described in Chapter 5. The V flag is tested by the branching instructions BVS (branch on overflow set, V = 1) and BVC (branch on overflow clear, V = 0). We will discuss branching instructions in Chapter 6. The overflow flag can be cleared to 0 by executing the instruction CLV (clear overflow flag; op-code = B8).

Decimal Mode Flag (D)

The decimal mode flag is bit 3 of the status register. If this flag is 0, then the arithmetic instructions ADC (add with carry) and SBC (subtract with carry) will perform binary, or hexadecimal, arithmetic. If the decimal mode flag, D, is set to 1, then the instructions ADC and SBC will perform decimal, or BCD (binary coded decimal), arithmetic. Examples of these two types of arithmetic will be illustrated in Chapter 5.

The D flag is set to 1 with the instruction SED (set decimal mode; op-code = F8) and is cleared to 0 with the instruction CLD (clear decimal mode; op-code = D8).

Break Flag (B)

The break flag is bit 4 of the status register. It is set to 1 when the BREAK instruction (a software interrupt; op-code = 00) is executed. The B flag is used by a program to determine if an interrupt resulted from a software interrupt (BREAK instruction) or from a hardware interrupt. Interrupts will be described in detail in Chapter 15.

Interrupt Disable Flag (I)

The interrupt disable flag is bit 2 of the status register. When it is set to 1, hardware interrupts entering the $\overline{\text{IRQ}}$ pin of the microprocessor are masked and the 6502 will not respond to the interrupt. When the I flag is cleared to 0, interrupts are unmasked and the 6502 will service hardware interrupts.

The I flag is set to 1 with the instruction SEI (set interrupt disable flag; op-code = 78) and is cleared to 0 with the instruction CLI (clear interrupt disable flag; op-code = 58). A detailed discussion of interrupts will be given in Chapter 15.

SHIFTING AND ROTATE INSTRUCTIONS

There are four instructions that allow you to move bits around in the accumulator or in any memory location. We will study these instructions by using accumulator A. Similar results can be obtained on the contents of any memory location by using some of the addressing modes described in Chapter 8.

Arithmetic Shift Left (ASL)

The ASL instruction with an op-code of 0A will cause the 8 bits in accumulator A to be shifted 1 bit to the left. The leftmost bit (bit 7) will be shifted into the carry bit. A 0 will be shifted into the rightmost bit (bit 0).

To see how this instruction works, store the hex value $AA in accumulator A by typing /RA AA, and store the op-code 0A in locations $300–$307 by typing

```
>300
/MH 0A 0A 0A 0A 0A 0A 0A 0A
```

Now move the position cursor back to location $300 and single step (CTRL S) eight times. You should observe the shifting of the bits in accumulator A, as shown in Figure 4.5. Note how the carry bit in the status register is changed each time you execute ASL.

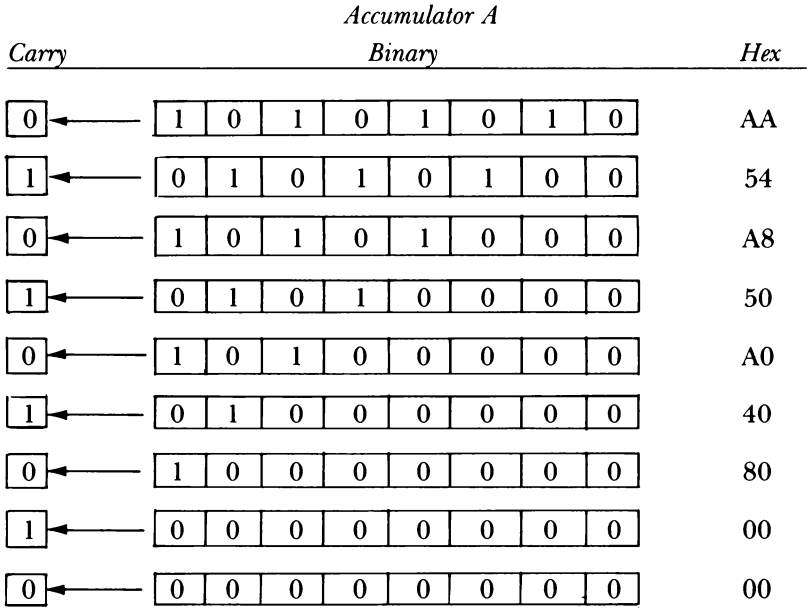


FIGURE 4.5. Result of executing ASL instruction eight times.

EXERCISE 4.2

Store the hex value $7B in accumulator A and execute the instruction ASL (0A) eight times. What is the value of the carry bit and the hex value in accumulator A after executing each instruction?

Logic Shift Right (LSR)

The LSR instruction with an op-code of 4A will cause the 8 bits in accumulator A to be shifted 1 bit to the right. The rightmost bit (bit 0) will be shifted into the carry bit. A 0 will be shifted into the leftmost bit (bit 7). A picture of what this instruction does is shown in Figure 4.6.

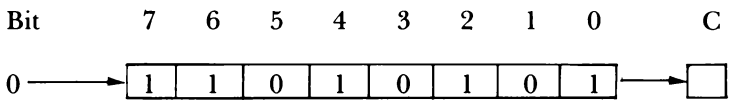


FIGURE 4.6. The LSR instruction (op-code = 4A).

EXERCISE 4.3

Store the hex value $D5 in accumulator A and execute the instruction LSR (4A) eight times. What is the value in the carry bit and the hex value in accumulator A after executing each instruction?

Rotate Left (ROL)

The rotate left instruction ROL differs from the arithmetic shift left instruction in that the carry bit is shifted into the rightmost bit rather than a 0, as shown in Figure 4.7. Each time that the instruction is executed, all bits are shifted 1 bit to the left. Bit 7 is shifted into the carry and the carry bit is shifted into bit 0.

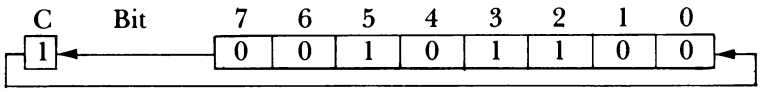


FIGURE 4.7. The ROL instruction (op-code = 2A).

EXERCISE 4.4

Store the hex value $2C in accumulator A and a 1 in the carry bit. (You can store a 1 in the carry bit by typing /RC 0F). Execute the instruction ROL (2A) eight times. What is the value in the carry bit and the hex value in accumulator A after executing each instruction?

Rotate Right (ROR)

The rotate right instruction ROR is just the opposite of rotate left. As shown in Figure 4.8, each bit in the accumulator is shifted 1 bit to the right. Bit 0 is shifted into the carry and the carry bit is shifted into bit 7.

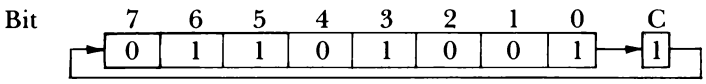


FIGURE 4.8. The ROR instruction (op-code = 6A).

EXERCISE 4.5

Store the hex value $69 in accumulator A and a 1 in the carry bit. Execute the instruction ROR (6A) eight times. What is the value in the carry bit and the hex value in accumulator A after executing each instruction?

INDEX REGISTERS X AND Y

The index registers X and Y are 8-bit registers that are used for several different purposes. They can be used in a manner similar to accumulator A for temporary storage when moving data to and from memory using the load and store instructions LDX, LDY, STX, and STY. The various addressing modes that can be used with these instructions are described in Chapter 8. Data can also be transferred between the index registers

and accumulator A using the instructions TAX, TXA, TAY, and TYA, shown in Table 4.1.

As an example, load the hex value $7D in the index register X by typing /RX 7D. Then load the op-codes 8A (TXA) and A8 (TAY) into memory locations $300 and $301 by typing

```
>300
/MH 8A A8
```

Now single step through these two instructions (CTRL S) and watch the 7D get transferred first to accumulator A and then from accumulator A to the index register Y.

Table 4.1 Transfer Instructions for the Index Registers

Instruction	*Op-Code*	*Meaning*	
TAX	AA	Transfer A to X	A → X
TXA	8A	Transfer X to A	X → A
TAY	A8	Transfer A to Y	A → Y
TYA	98	Transfer Y to A	Y → A

The index registers are often used as 8-bit counters. This is done by loading a value into the index register and then incrementing or decrementing this value using the instructions INX, DEX, INY, or DEY, shown in Table 4.2.

As an example, load the value $04 into index register X by typing /RX 04 and then load the op-code CA (DEX) in memory locations $300 through $304 by typing

```
>300
/MH CA CA CA CA CA
```

Table 4.2 Incrementing and Decrementing Instructions for the Index Registers

Instruction	*Op-Code*	*Meaning*
INX	E8	Increment X, X = X + 1
DEX	CA	Decrement X, X = X − 1
INY	C8	Increment Y, Y = Y + 1
DEY	88	Decrement Y, Y = Y − 1

Note that when you single step through these five instructions the value in the index register X will be decremented by 1 each time the op-code

CA is executed. When the fourth CA is executed, the value in X will become 0 and the Z-bit in the status register will be set to 1. When the fifth CA is executed the value in X will become FF and the N-bit in the status register will be set to 1, because bit 7 in X is now set and the contents of X can be interpreted as a negative number. As we will see in the next chapter, FF is the two's complement representation of -1.

The index registers X and Y play important roles in the various indexed addressing modes to be described in Chapter 8. These indexed addressing modes are particularly useful for accessing data that are stored in memory in the form of a table.

STACK POINTER (SP)

The stack is a region of memory that is set aside for storing temporary data. In 6502 systems the stack is always on page 1 of memory—that is, between memory locations $0100 and $01FF. The stack pointer, SP, is an 8-bit register that contains the least significant byte of the address corresponding to the top of the stack. The most significant byte of this address is always 01. The TUTOR monitor displays the contents of the stack pointer at the upper-right-hand corner of the screen. This value is set to 7F when you start the TUTOR program. You can change this to another value such as 60 by typing /RS 60. Try it.

The two instructions TSX and TXS, shown in Table 4.3, allow you to transfer bytes between the index register X and the stack pointer S. At the beginning of a program it is a good idea to set the stack pointer to some known location. For example, to set the stack pointer to 50, execute the following instructions:

```
300   A2 50   LDX #$50
302   9A      TXS
```

Enter the 3 bytes A2 50 9A starting at location $300; then single step through these two instructions while watching the contents of X and SP.

Table 4.3 Transfer Instructions for the Stack Pointer

Instruction	*Op-Code*	*Meaning*
TSX	BA	Transfer SP to X SP → X
TXS	9A	Transfer X to SP X → SP

An important use of the stack is to save return address information when a subroutine is called. Data in accumulator A can be stored on the stack

by using the instruction PHA (push A). Data can be removed from the stack by using the instruction PLA (pull A). Examples using these instructions together with a discussion of subroutines will be given in Chapter 7, where the operation of the stack will be described in detail.

EXERCISE 4.6

Single step through a program that loads the hex value $2C into accumulator A, transfers the value to index register X, increments the value to $30, and then stores this result in the stack pointer.

5

6502 Arithmetic

The 6502 microprocessor supports two types of arithmetic: binary and decimal. The type of arithmetic performed depends on the D flag in the status register. If the D flag is 0, the 6502 will perform binary arithmetic. If the D flag is 1, the 6502 will perform decimal arithmetic. We will describe these two different types of arithmetic separately.

BINARY ARITHMETIC

If you want to have the 6502 microprocessor perform binary arithmetic, you must make sure that the D flag in the status register is set to 0 by executing the instruction CLD (clear decimal mode; op-code = D8) at the beginning of the program. Once this is done, the instructions ADC (add with carry) and SBC (subtract with carry) will perform binary addition and subtraction.

Binary Addition

The addition of binary numbers is carried out bit by bit starting with the least significant bit (the rightmost bit) according to Table 5.1.

Table 5.1 Binary Addition

Carry (in)	*A*	*B*	*A + B + Carry*	*Carry (out)*
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

The only addition instruction in the 6502 is add with carry (ADC). This means that before you begin an addition you must clear the carry bit by executing the CLC instruction (op-code = 18).

Consider the following binary addition:

	Binary	Hex	Decimal
Carry	[0] 01100010		
A	00110101	35	53
B	00011001	19	25
	01001110	4E	78

Note that the initial carry bit is 0 and the final carry bit (0) is shown in the box. This is the carry bit that is displayed in the status register after the instruction ADC is executed. The intermediate carry bits are determined according to Table 5.1.

The binary values of A and B shown in the addition example are equivalent to the hexadecimal numbers 35 and 19. These can be added directly (in hexadecimal) as shown. The equivalent decimal addition (53 + 25 = 78) is also shown.

The program shown in Figure 5.1 will perform this addition. Key in this program by typing

```
>300
/MH D8 18 A9 35 69 19
```

```
300   D8      CLD        Clear decimal mode
301   18      CLC        Clear carry
302   A9 35   LDA #$35   A = $35
304   69 19   ADC #$19   A = A + $19 + C
```

FIGURE 5.1. Program to add $35 and $19.

Single step through this program and observe the contents of accumulator A and the status register.

Negative Numbers

An 8-bit number can represent one of 256 (2^8) possible values between 0 and 255. However, we also need to represent negative numbers. The leftmost bit in a byte (bit 7) is the *sign bit.* If this bit is 0 the number is positive; if it is 1, the number is negative. In the 6502, however (and in most computers today), when bit 7 is 1, the magnitude of the negative number is *not* given by the binary value of the remaining 7 bits in the byte. Rather, a two's complement representation of negative numbers is used. The reason for this is that the same circuitry—an adder—can be used for both addition and subtraction.

The idea of being able to subtract by adding can be seen by an example using decimal (base 10) numbers. Suppose that you want to subtract 35 from 73. The answer is 38. You can obtain this result by taking the ten's complement of 35 (this is 65, the number you have to add to 35 to get 100) and adding it to 73. The result is 138, as shown in Figure 5.2. If you ignore the leading 1 (the carry), the remaining value, 38, is the correct answer.

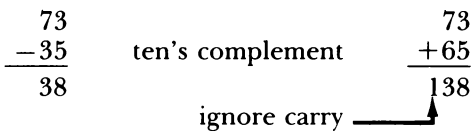


FIGURE 5.2. Decimal subtraction can be done by taking ten's complement of the subtrahend and adding.

In binary arithmetic, negative numbers are stored in their two's complement form. You can find the two's complement of a binary number in several ways. Note that the ten's complement of 35 can be found by subtracting 35 from 99 (this gives the nine's complement) and then adding 1. That is,

```
  99
 −35
  64
 + 1
  65
```

The two's complement of the 8-bit binary number 01001101 is the 8-bit binary number you must add to this number to obtain 100000000. You

can find it by subtracting the number from 11111111 and adding 1. Note that subtracting an 8-bit binary number from 11111111 (called the one's complement) is equivalent to complementing each bit in the byte; that is, each 1 is changed to a 0, and each 0 is changed to a 1. Therefore, the one's complement of 01001101 is 10110010 and the two's complement of 01001101 is

	01001101
one's complement =	10110010
add	1
two's complement =	10110011

There is an easier way to take the two's complement of a binary number. You just start at the rightmost bit and copy down all bits until you have copied down the first 1. Then complement (that is, change from 1 to 0, or 0 to 1) all of the remaining bits. For example,

```
complement remaining bits——┐┌copy this first 1
                     0 1 0 0 1 1 0 1
two's complement =   1 0 1 1 0 0 1 1
```

As a second example,

```
complement remaining bits┐┌—— copy all bits to first 1
                     0 1 0 1 1 0 0 0
two's complement =   1 0 1 0 1 0 0 0
```

Verify that if you add the 8-bit binary numbers given in these examples to their two's complement value, you obtain 100000000.

An 8-bit byte can contain positive values between 00000000 and 01111111—that is, between \$00 and \$7F. This corresponds to decimal values between 0 and 127. A byte in which bit 7 is a 1 is interpreted as a negative number whose magnitude can be found by taking the two's complement. For example, how is -75_{10} stored in the computer? First write down the binary or hex value of the number, \$4B, as shown in Figure 5.3. Then take its two's complement. The value of -75_{10} is therefore stored in the computer as \$B5. Note that if you take the two's complement of a positive number between 0 and \$7F, the result will always have bit 7 set to 1.

	75_{10}	= \$4B	= 01001011
two's complement =	-75_{10}	= \$B5	= 10110101
two's complement of	\$B5	= \$4B	= 01001011

FIGURE 5.3. The negative of a binary number is found by taking the two's complement.

You can always find the magnitude of a given negative number (with bit 7 set) by taking the two's complement. For example, the two's complement of \$B5 (-75_{10}) is \$4B ($+75_{10}$), as shown in Figure 5.3.

Note that the two's complement of \$01 is \$FF and the two's complement of \$80 is \$80, as shown in Figure 5.4. This last example shows that signed 8-bit binary numbers "wrap around" at \$80. That is, the largest positive number is \$7F = 127_{10} and the smallest negative number (largest magnitude) is \$80 = -128_{10}. This is shown in Table 5.2.

$$
\begin{aligned}
1_{10} &= \$01 = 00000001 \\
\text{two's complement} \quad = -1_{10} &= \$FF = 11111111 \\
128_{10} &= \$80 = 10000000 \\
\text{two's complement} = -128 &= \$80 = 10000000
\end{aligned}
$$

FIGURE 5.4. Negative numbers can range between \$FF (−1) and \$80 (−128).

Table 5.2 Positive and Negative Binary Numbers

Signed Decimal	*Hex*	*Binary*	*Unsigned Decimal*
− 128	80	10000000	128
− 127	81	10000001	129
− 126	82	10000010	130
•		•	•
•		•	•
•		•	•
− 3	FD	11111101	253
− 2	FE	11111110	254
− 1	FF	11111111	255
0	00	00000000	0
1	01	00000001	1
2	02	00000010	2
3	03	00000011	3
•		•	•
•		•	•
•		•	•
125	7D	01111101	125
126	7E	01111110	126
127	7F	01111111	127

Table 5.2 also shows that the hex values between \$80 and \$FF can be interpreted *either* as negative numbers between − 128 and − 1 *or* as positive numbers between 128 and 255. The 6502 sometimes treats these values as negative numbers and sometimes treats them as positive values. It

is up to you as the programmer to make sure you know whether a particular byte is being treated as a signed number or an unsigned number.

EXERCISE 5.1

Find the hex values for the following decimal numbers:

1. − 7
2. − 101
3. − 68
4. − 25
5. − 120
6. − 5

EXERCISE 5.2

The following hex values correspond to what negative decimal numbers?

1. $CD
2. $F3
3. $E2
4. $85
5. $99
6. $AB

Carry and Overflow

The program shown in Figure 5.1 adds the hexadecimal numbers $35 and $19. Enter this program at memory location $300 using the TUTOR monitor. The result of this addition is shown in Figure 5.5. Single step through this program. Note that when the addition instruction at location $304 is executed, the carry flag C and the overflow flag V are both cleared to 0.

Decimal	Hex	Binary
53	35	00110101
+ 25	19	00011001
78	4E	C [0] 01001110
		V [0]

FIGURE 5.5. There is no carry from bit 6 to bit 7 and no carry from bit 7 to C.

Decimal	Hex	Binary
53	35	00110101
+ 91	5B	01011011
144	90	C [0] 10010000
		V [1]

FIGURE 5.6. An overflow occurs (V=1) when there is a carry from bit 6 to bit 7 and no carry from bit 7 to C.

Now modify this program by changing the $19 in location $305 to $5B. This is equivalent to adding the decimal numbers 53 and 91, as shown in Figure 5.6. Single step through this new program and note that when the addition instruction at location $304 is executed, the carry flag C is cleared to 0 but the overflow Flag V is set to 1. There is an overflow because the answer $90 is really the *negative* (two's complement) value -112_{10}. (Verify this.) Although $90 is equivalent to the positive value 144_{10} when the result is thought of as an 8-bit unsigned number, the overflow flag V in the status register always thinks of the result as a signed number between − 128 and + 127. If the correct result is outside this range (144 in Figure 5.6), then the overflow flag V is set to 1.

Now modify the program again by changing the $5B in location $305 to $D3. The hex value $D3 represents the negative decimal number − 45, as can be seen by taking the two's complement of $D3:

$$\$D3 = 11010011$$
$$\text{two's complement} = 00101101 = \$2D = 45_{10}$$

Therefore, adding $D3 to $35 is the same as subtracting 45_{10} from 53_{10}, as shown in Figure 5.7. Single step through this program and note that when the addition instruction at location $304 is executed, the carry flag C is set to 1 but the overflow flag V is cleared to 0. There is no overflow because the binary addition result ($08) is correct. The result will always be correct (and therefore V will be cleared to 0) if there is a carry from bit 7 to C *and* a carry from bit 6 to bit 7.

Decimal	Hex	Binary
53	35	00110101
− 45	D3	11010011
8	1 08	C[1]00001000
	ignore carry	V[0]

FIGURE 5.7. The overflow flag V is cleared to 0 when there is a carry from bit 6 to bit 7 *and* a carry from bit 7 to C.

Finally, change the value in location $303 from $35 to $9E. The hex value $9E is equivalent to the negative decimal value − 98:

$$\$9E = 10011110$$
$$\text{two's complement} = 01100010 = \$62 = 98_{10}$$

Figure 5.8 shows that adding − 98 and − 45 produces the decimal value − 143 and the hex value $71, which is incorrect. The V flag and the carry are both set to 1. Verify this by single stepping through the program.

Decimal	Hex	Binary
− 98	9E	10011110
− 45	D3	11010011
− 143	1 71	C[1]01110001
	ignore carry	V[1]

FIGURE 5.8. The overflow flag V is set to 1 when there is a carry from bit 7 to C but no carry from bit 6 to bit 7.

The results illustrated in Figures 5.5 to 5.8 show that the overflow flag V is set to 1 if there is a carry from bit 6 to bit 7 with no carry from bit 7 to C (Figure 5.6) *or* if there is a carry from bit 7 to C with no carry from bit 6 to bit 7 (Figure 5.8). If there is no carry from bit 6 to bit 7 *and* no carry from bit 7 to C (Figure 5.5), *or* if there is a carry from bit 6 to bit 7 *and* a carry from bit 7 to C (Figure 5.7), then there is no overflow and V = 0. This can be summarized by saying that the overflow flag V is the *exclusive-or* (+) of a carry from bit 6 to bit 7 and a carry from bit 7 to C. That is,

$$V = \text{carry from bit 6 to bit 7} \oplus \text{carry from bit 7 to C}$$

where A $\oplus$ B is defined according to Table 5.3.

Table 5.3 Exclusive-or ($\oplus$) operation

A	*B*	*A* $\oplus$ *B*
0	0	0
0	1	1
1	0	1
1	1	0

Binary Subtraction

The subtraction of binary numbers can be carried out bit by bit starting with the least significant bit according to Table 5.4. An example is shown in Figure 5.9. Note how a borrow may be required when subtracting bits. The only subtraction instruction in the 6502 is subtract with carry, SBC. This is actually a subtract with borrow, where the borrow is the complement of the carry bit C. Thus, the instruction SBC #$6F performs the operation A − $6F − $\overline{C}$.

Table 5.4 Binary Subtraction

Carry	*Borrow*=$\overline{C}$	*A*	*B*	*A*−*B*−$\overline{C}$	*Borrow*=$\overline{C}$	*Carry*
0	1	0	0	1	1	0
0	1	0	1	0	1	0
0	1	1	0	0	0	1
0	1	1	1	1	1	0
1	0	0	0	0	0	1
1	0	0	1	1	1	0
1	0	1	0	1	0	1
1	0	1	1	0	0	1

Decimal	Hex	Binary
Borrow 0	Borrow 0 1	Borrow 0 1 111
181	B5	10110101
− 111	− 6F	− 01101111
70	46	01000110

FIGURE 5.9. Example of binary subtraction.

Borrow 0		
B5	B5	10110101
− 6F	+ 91	10010001
46	C[1] 46	C[1] 01000110

FIGURE 5.10. Binary subtraction is equivalent to taking the two's complement of the subtrahend and adding.

The subtraction shown in Figure 5.9 is equivalent to taking the two's complement of the subtrahend and adding, as shown in Figure 5.10. Note that this addition causes the carry bit C to be set, which corresponds to no borrow ($\overline{C} = 0$).

Because the instruction SBC always subtracts $\overline{C}$ (the borrow) from the result, it is necessary to set the carry flag to 1 (so that $\overline{C}$ will equal 0), using the instruction SEC before executing SBC.

```
300   D8      CLD          Clear decimal mode
301   38      SEC          Set carry
302   A9 B5   LDA #$B5     A = $B5
304   E9 6F   SBC #$19     A = A − $6F − C
```

FIGURE 5.11. Program to subtract $6F from $B5.

The subtraction illustrated in Figures 5.9 and 5.10 can be performed by executing the instructions shown in Figure 5.11. Key in this program by typing

```
> 300
/MH D8 38 A9 B5 E9 6F
```

Single step through this program and observe the contents of accumulator A and the status register. Note that after the subtraction the carry bit is set, indicating *no borrow*. The carry will always be set if the magnitude (0 –255) of the minuend ($B5 in Figure 5.9) is larger than or equal to the magnitude of the subtrahend ($6F in Figure 5.9). If the magnitude (0–255) of the minuend is *less than* the magnitude of the subtrahend, the carry flag will be cleared to 0 (corresponding to a borrow).

The overflow flag V only has meaning when you are subtracting signed (two's complement) numbers. As in addition, the overflow flag V is set to 1 when the result is outside the range − 128 through +127.

EXERCISE 5.3

Modify the program in Figure 5.11 to perform the following subtractions. In each case explain the answer in accumulator A and the value of the carry flag C and the overflow flag V.

1. $73 − $A1
2. $D3 − $47
3. $BB − $F2
4. $E1 − $C3

DECIMAL ARITHMETIC

Although computers add and subtract binary numbers, people using the computers are more used to dealing with decimal numbers. For this reason, decimal numbers are normally entered through the keyboard and displayed on the screen. This means that the computer must convert a decimal number entered from the keyboard to a binary number, perform a calculation, and then convert the binary result to a decimal number before displaying it on the screen.

An alternative is to do the calculation in decimal, thus avoiding the conversion to binary. The 6502 microprocessor allows such calculations by operating directly on binary coded decimal (BCD) digits. A BCD digit is one of the decimal digits 0–9. These digits are coded using the 4-bit binary equivalent representations 0000–1001. The 4-bit combinations corresponding to the hex digits A–F are not allowed in BCD numbers. An 8-bit

byte can contain two BCD digits. Thus, the decimal number 35 is stored in packed BCD format as \$35 = 00110101. Note that as a BCD number this is interpreted as 35_{10} and *not* as $35_{16} = 53_{10}$.

If the decimal mode D flag in the status register is set to 1, then the add instruction, ADC, will perform BCD addition and the subtract instruction, SBC, will perform BCD subtraction.

BCD Addition

An example of the difference between binary and decimal addition is shown in Figure 5.12. Note that the hex values \$35 and \$47 are used in both cases. The only difference is the value of the D flag. If D = 1, the hex values \$35 and \$47 are interpreted as decimal numbers rather than binary numbers.

Binary, D = 0		Decimal, D = 1	
\$35	00110101	\$35	00110101
+ \$47	01000111	+ \$47	01000111
\$7C	01111100	\$82	10000010

FIGURE 5.12. Binary and decimal addition.

To see this, enter the program shown in Figure 5.13 and single step through the four instructions. When using the decimal mode, the carry will be set when the result of the decimal addition exceeds 99.

Note that packed BCD numbers use all 8 bits in a byte (4 bits for each of two digits). Therefore, no sign bit is associated with BCD numbers. You must keep track of the sign of BCD numbers separately.

```
300    F8      SED          Set decimal mode
301    18      CLC          Clear carry
302    A9 35   LDA #$35     A = 35
304    69 47   ADC #$47     A = A + 47 + C
```

FIGURE 5.13. Program to add BCD numbers.

EXERCISE 5.4

Modify the program in Figure 5.13 to perform the following decimal additions. In each case indicate the answer in accumulator A and the value of the carry flag C.

1. 49 + 34
2. 73 + 47
3. 29 + 36
4. 55 + 69

BCD Subtraction

An example of the difference between binary and decimal subtraction is shown in Figure 5.14.

Binary, D = 0		Decimal, D = 1	
$52	01010010	$52	01010010
− $25	00100101	− $25	00100101
$2D	00101101	$27	00100111

FIGURE 5.14. Binary and decimal subtraction.

The program shown in Figure 5.15 will subtract the decimal number 25 from 52. Type in and single step through this program. BCD subtraction will set the carry flag when the decimal value of the minuend is less than the decimal value of the subtrahend.

```
300   F8      SED          Set decimal mode
301   38      SEC          Set carry
302   A9 52   LDA #$52     A = 52
304   E9 25   SBC #$25     A = A - 25 - C
```

FIGURE 5.15. Program to subtract BCD numbers.

EXERCISE 5.5

Modify the program in Figure 5.15 to perform the following decimal subtractions. In each case indicate the answer in accumulator A and the value of the carry flag C.

1. 89 − 35
2. 63 − 27
3. 46 − 63
4. 23 − 47

Branching Instructions

A computer program achieves its apparent power by being able to conditionally branch to different parts of a program. The 6502 microprocessor uses branching instructions for this purpose. Each branching instruction can cause a branch in the program to occur, depending upon the state of one of the bits in the status register. In this chapter you will learn

1. the 6502 branching instructions
2. how to calculate the branching offset
3. how to have the TUTOR calculate the branching offset
4. to use the BNE and BEQ instructions to branch on the state of the zero flag Z
5. to use the BPL and BMI instructions to branch on the state of the negative flag N
6. to use the BCC and BCS instructions to branch on the state of the carry flag C

THE 6502 CONDITIONAL BRANCHING INSTRUCTIONS

The 6502 has eight conditional branching instructions, shown in Table 6.1. The branch test for each instruction is a test of the state of one of the status register flags. For example, the branching instruction BEQ will cause a branch in the program if the Z flag in the status register is 1. This

Table 6.1 Branching Instructions

Operation	*Mnemonic*	*Op-Code*	*Branch Test*
Branch if equal 0	BEQ	F0	Z = 1
Branch if not equal 0	BNE	D0	Z = 0
Branch if plus	BPL	10	N = 0
Branch if minus	BMI	30	N = 1
Branch if carry clear	BCC	90	C = 0
Branch if carry set	BCS	B0	C = 1
Branch if overflow clear	BVC	50	V = 0
Branch if overflow set	BVS	70	V = 1

All branching instructions are 2 bytes long and use *relative addressing*. Branching instructions take three machine cycles to execute if no page boundary is crossed, or four cycles if a page boundary is crossed.

will be the case if the result of the previous instruction produced a result of 0.

All branching instructions are 2 bytes long. The first byte is the op-code, whose values for the eight branching instructions are given in Table 6.1. The second byte of the instruction is the relative offset of the branch destination. This is the two's complement number that must be added to the value of the program counter + 2 to obtain the address of the instruction to be executed if the branch test is *true*. If the branch test is *false*, then the instruction following the branching instruction is executed. This is illustrated in Figure 6.1. Note that if Z = 0 when the BEQ instruction at location $0312 is executed, the next instruction that is executed will be the one at location $0314. On the other hand, if Z = 1 when the BEQ instruction is executed, the program will branch to the address formed by adding the offset (06) to the address of the next instruction ($0314)—that is, to location 031A = $0314 + $06.

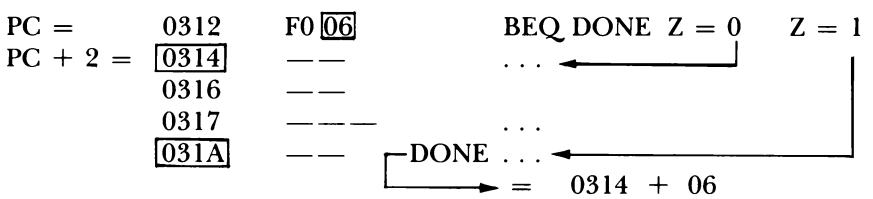


FIGURE 6.1. The offset in a branching instruction is added to the value of the program counter + 2 to obtain the destination address of the branch.

If a branching instruction branches backward in memory, the offset must be negative. It is just the two's complement of the number of bytes between the address of the next instruction (PC + 2) and the branch destination address. Note that since the branching offset is a single 8-bit byte, a branching instruction can only branch forward a maximum of 127 bytes

($7F) and backward a maximum of − 128 bytes ($80). The counting of these bytes always begins at the address of the instruction *following* the branching instruction. These offsets can be calculated either by hand or automatically by the TUTOR monitor.

CALCULATING BRANCHING OFFSETS

Suppose that a branching instruction is to branch backward − 8 bytes from the address of the next instruction. Since − 8 is represented as a two's complement hexadecimal number $F8, the branching offset will be F8, as shown in Figure 6.2. This value can be calculated by subtracting the address PC + 2 ($0314) from the destination address ($030C), as shown in Figure 6.2. Note that this subtraction is done by taking the two's complement of 0314 (PC + 2) and adding the result to 030C (destination address). The result, FFF8, is the 16-bit hexadecimal representation of -8_{10}. When a two's complement, 8-bit hexadecimal number such as F8 is stored as a 16-bit number, the sign bit (1 in this case) is extended to the left through the high-order byte. Thus, F8 and FFF8 both represent the negative number -8_{10}.

```
                                                        FFFF
          030C —        LOOP —          PC + 2 =      0314
          030D ——         —                           FCEB
          030F ——         —                         + 1
                                        two's compl. = FCEC

PC      = 0312 D0 F8      BNE     Dest. addr. =     030C
PC + 2  = 0314 ——         LOOP                      FCEC
          0316 —                                    FFF8
                                                      ↑
                                                    offset
```

FIGURE 6.2. Negative branches can be found by subtracting the address PC + 2 from the destination address.

Calculating the offset of a branching instruction incorrectly is a common mistake made when writing machine language programs. To avoid such mistakes, the TUTOR monitor will calculate these offsets for you. When you enter a machine language program with the TUTOR (by using /MH) and you come to the location of a branch offset, just type 00. This will leave a byte where the offset is to go. Then go back to each of these offset locations and type /O. The command line will display

OFFSET: DESTINATION ADDRESS

Enter the destination address and the correct offset will automatically be inserted at the current position cursor location (that is, at the location of the offset byte).

Try this by going to location $313 (>313). Then type /O 30C. The offset F8 should be inserted at $313, as shown in Figure 6.2. If you now type /O 31A, the offset 06 will be inserted at $313, as shown in Figure 6.1.

BRANCHING EXAMPLES

The following short examples will illustrate branching on the state of the Z, N, and C flags.

Branching on the Zero Flag Z

Type in the program shown in Figure 6.3, starting at location $300. You should verify the offsets FD (at location $304) and F9 (at location $306) by using the /O offset calculation feature of the TUTOR described previously.

```
300    A2 03    LOOP1   LDX  #$03
302    CA       LOOP2   DEX
303    D0 FD            BNE  LOOP2
305    F0 F9            BEQ  LOOP1
```

FIGURE 6.3. BNE and BEQ branch on Z = 0 and Z = 1, respectively.

Now single step through this program, starting at location $300. After you have executed the instruction DEX (at $302) the first time, the value of X will be $02 and the value of the zero flag Z will be 0. Therefore, the BNE instruction at location $303 will branch back to location $302 and execute DEX again. The value of X will now be $01 and the Z flag will still be 0. Therefore, the BNE instruction will branch back to location $302 again. This time the DEX instruction will cause the value of the index register X to go to 0. This will cause the Z flag to be set to 1. The test Z = 0 of the BNE instruction will fail so that another branch cannot occur. The next instruction at location $305 will therefore be executed. This is a BEQ instruction that will branch if Z = 1. But the Z flag will be equal to 1 (otherwise, the BNE instruction would have branched); the program will then branch to location $300 and you can single step through the program again. Single step through this program several times, observing the value of the Z flag and the contents of the X index register.

Branching on the Negative Flag N

The program shown in Figure 6.4 will test the branching instructions BPL and BMI. Type in this program and single step through it. The value in index register Y is set to $7D at location $300 and then incremented by 1 (to $7E) at location $302. The N flag will be 0 ($7E is a positive number) so that the BPL instruction will branch back to location $302.

```
300     A0 7D    LOOP1   LDY  #$7D
302     C8       LOOP2   INY
303     10 FD            BPL  LOOP2
305     30 F9            BMI  LOOP1
```

FIGURE 6.4. BPL and BMI branch on N = 0 and N = 1, respectively.

The index register Y will then be incremented to $7F (still positive) so that the BPL instruction will branch back again to the INY instruction. This time Y will be incremented to $80, which is a negative number (-128_{10}), because bit 7 is set to 1. This will cause the negative flag N in the status register to be set to 1 so that the BPL test (Z = 0) will fail. The BMI instruction at location $305 will then be executed; this will always (because Z = 1) branch back to location $300. Single step through this program several times and observe the value of the N flag and the contents of the Y index register.

Branching on the Carry Flag C

The example given in Figure 6.3 decrements the X index register until it becomes 0. The example given in Figure 6.4 increments the Y index register until it becomes negative (equal to $80). Suppose that you wanted to increment the Y index register as long as it was less than a particular value, say $2B. The program shown in Figure 6.5 will do this. Type in this program and single step through it.

The CPY (compare Y) instruction at location $303 will subtract the value $2B from the current value of Y. Recall from Chapter 5 that the carry flag, C, will be set to 1 if the magnitude of Y (considered to be an 8-bit positive number from 0 to 255_{10}) is greater than or equal to $2B. If Y is less than $2B, the carry flag will be cleared to 0. Thus, the BCC (branch on carry clear) branch instruction at location $305 can be thought of as a "branch if less than" instruction. That is, if Y is less than $2B a branch will occur. The BCS (branch on carry set) instruction at location $307 will always branch because the carry flag will have to be set to get to the instruction.

```
300     A0 28    LOOP1   LDY  #$28
302     C8       LOOP2   INY
303     C0 2B            CPY  #$2B
305     90 FB            BCC  LOOP2
307     B0 F7            BCS  LOOP1
```

FIGURE 6.5. BCC and BCS branch on C = 0 and C = 1, respectively.

Single step through this program several times and observe the value of the C flag and the contents of the Y index register.

EXERCISE 6.1

Type in the following program and single step through it several times. Explain how each instruction affects the contents of index register X and the value of the carry flag C.

```
300     A2 19    LOOP1   LDX  #$19
302     CA       LOOP2   DEX
303     E0 16            CPX  #$16
305     B0 FB            BCS  LOOP2
307     90 F7            BCC  LOOP1
```

EXERCISE 6.2

Type in the following program and single step through it several times. Explain how each instruction affects the contents of accumulator A and the value of the overflow flag V.

```
300     18       LOOP1   CLC
301     A9 7A            LDA  #$7A
303     69 02    LOOP2   ADC  #$02
305     50 FC            BVC  LOOP2
307     70 F7            BVS  LOOP1
```

7

The Stack and Subroutines

The stack pointer is one of the 8-bit registers in the 6502 microprocessor that was described in Chapter 4. In this chapter you will learn

1. how a stack works
2. the 6502 push and pull instructions
3. to use the jump to subroutine (JSR) and return from subroutine (RTS) instructions
4. to write a *delay* subroutine
5. to use the Apple II speaker
6. how to set breakpoints with the TUTOR
7. how to execute programs with the TUTOR
8. to read the Apple II keyboard

THE STACK

The stack is a group of memory locations in which temporary data can be stored. A stack is different from any other collection of memory locations in that data can only be put on and taken from the *top* of the stack. The process is similar to stacking dinner plates on top of one another, where the last plate put on the stack is always the first one removed from it. We sometimes refer to this as a *last in–first out* or LIFO stack.

The memory location corresponding to the top of the stack (the next *empty* location) is stored in the stack pointer. The 6502 always uses page 1 for the stack; therefore, the most significant byte of the stack pointer address is 01. The 8-bit stack pointer of the 6502 contains the least significant byte of the address of the top of the stack. For example, if the stack pointer contains $7F, the address of the top of the stack is $017F.

When data are put on the stack, the stack pointer is *decremented.* This means that the stack grows *backward* in memory. As data values are put on the stack they are put into memory locations with lower addresses. Data can be put on and taken off the stack using *push* and *pull* instructions.

Push and Pull Instructions

The push and pull instructions of the 6502 are given in Table 7.1. Note that data can be transferred to and from the stack via the accumulator or the status register. To test these instructions, use the /R command to store a value of $11 in accumulator A, $22 in index register X, $33 in index register Y, $75 in the status register, and $7F in the stack pointer. Then enter the program shown in Figure 7.1, starting at location $300. Single step through this program and watch the stack at the right of the screen. Figure 7.2 shows what the screen will look like after executing the PHP statement at location $305. Note that the value of the stack pointer has been decremented from $7F to $7B.

Table 7.1 Push and Pull Instructions

Operation	*Mnemonic*	*Op-Code*
Push accumulator on stack	PHA	48
Pull accumulator from stack	PLA	68
Push processor status register on stack	PHP	08
Pull processor status register from stack	PLP	28

```
300  48  PHA
301  8A  TXA
302  48  PHA
303  98  TYA
304  48  PHA
305  08  PHP
306  68  PLA
307  68  PLA
308  68  PLA
309  28  PLP
```

FIGURE 7.1. Test program for demonstrating the push and pull instructions.

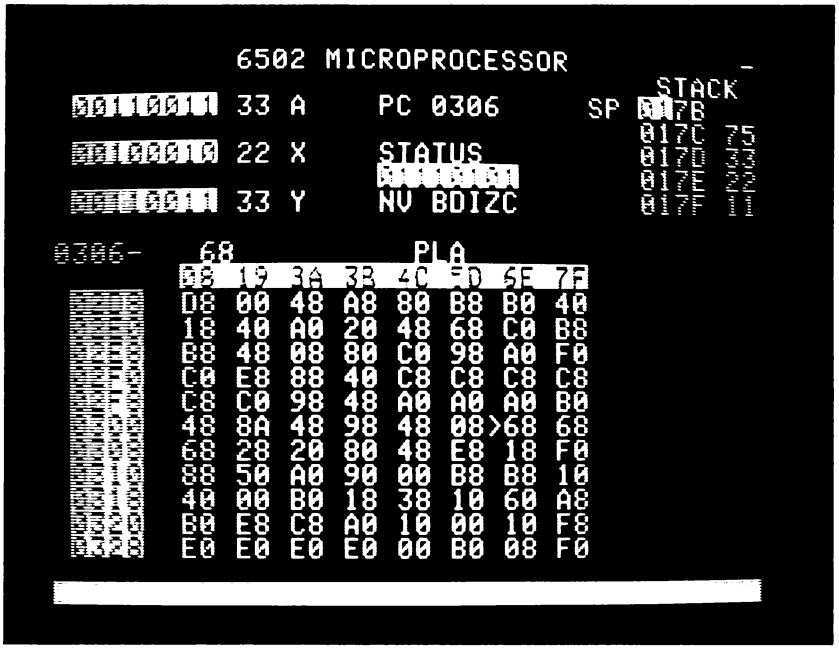


FIGURE 7.2. Screen display after pushing four values on the stack.

A *push* instruction (PHA or PHP) stores a value in a memory location whose address is given by the stack pointer, and then decrements the stack pointer by 1. A *pull* instruction (PLA or PLP) increments the stack pointer by 1, and then loads a value from the memory location pointed to by the stack pointer.

SUBROUTINES

A *subroutine* is a segment of code that is normally written to perform a particular function or task. A subroutine is called by executing the *jump to subroutine* (JSR) instruction. A subroutine is exited by executing the *return from subroutine* (RTS) instruction. This will cause the program to return to the instruction following the JSR instruction that called the subroutine.

The computer knows where to go when an RTS instruction is executed because it stored a return address on the stack when the JSR instruction was executed. To see how this works, key in all of the instructions shown in Figure 7.3.

The instruction at location $300 is JSR $308. The machine language version of this instruction is 20 08 03. Note that the address $0308 is stored with the low byte (08) first, followed by the high byte (03). The 6502 always stores addresses in this low-byte, high-byte format. If you sin-

```
300    20 08 03    JSR $308
303    20 10 03    JSR $310
306    00          BRK

308    20 10 03    JSR $310
30B    60          RTS

310    60          RTS
```

FIGURE 7.3. The JSR and RTS instructions.

gle step this instruction the program will jump to location $308 and the address $0302 will be pushed on the stack, as shown in Figure 7.4. Note that the high-order byte of the address $03 is pushed on the stack first and the low-order byte $02 is pushed on the stack last.

The address of the next instruction is $303. This is one more than the address stored on the stack. When an RTS instruction is executed it pulls the top address from the stack, adds 1 to it, and puts it in the program counter. This will cause the program to return to the instruction following the JSR instruction that called the subroutine.

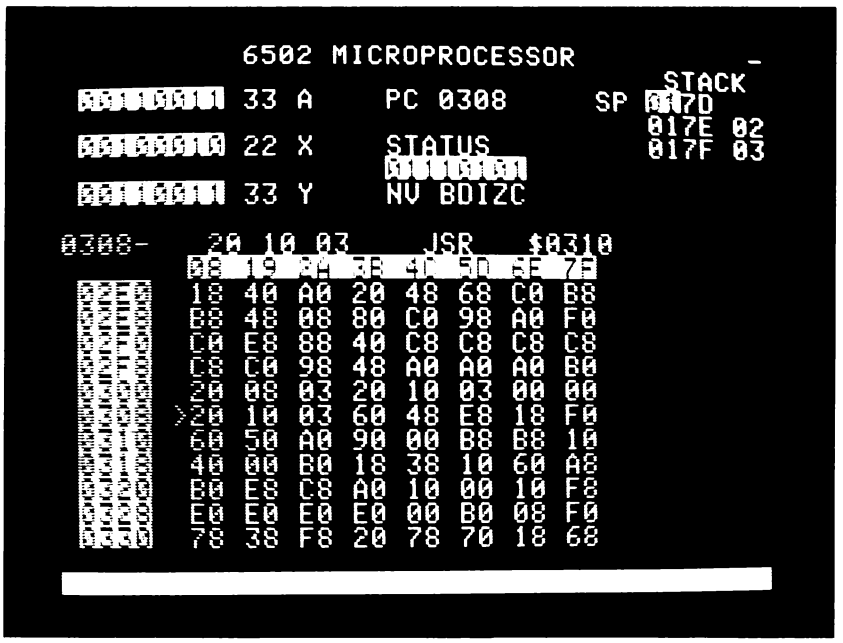


FIGURE 7.4. Screen display after executing JSR $308 at location $300.

The first instruction of the subroutine at location $308 is another JSR instruction that jumps to location $310. If you single step this instruction, the screen display will be as shown in Figure 7.5. Note that the program jumps to location $310 and the return address (minus 1), $030A, is pushed on the stack.

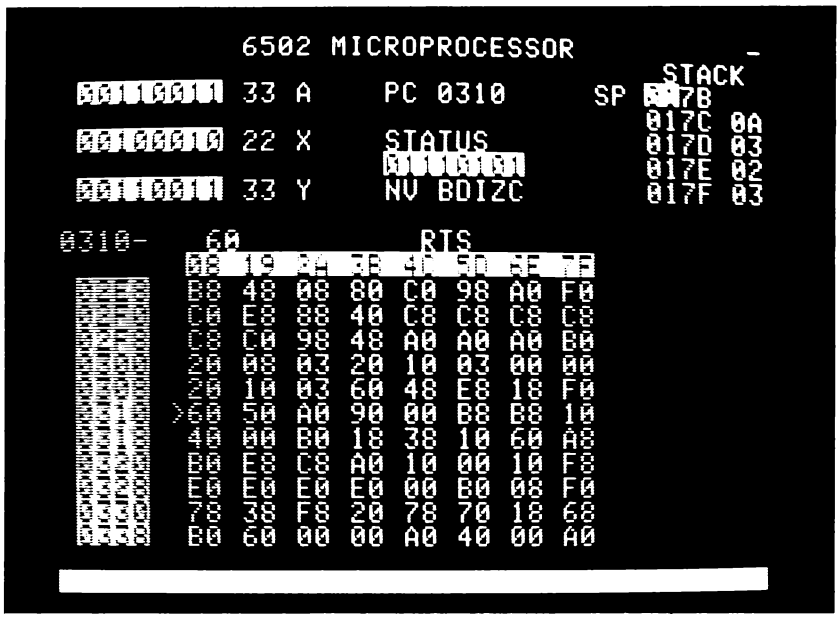


FIGURE 7.5. Screen display after executing JSR $310 at location $308.

The instruction at location $310 is RTS. If you single step this instruction the program will return to location $30B as shown in Figure 7.6. Note that the most recent return address (minus 1) has been pulled from the stack.

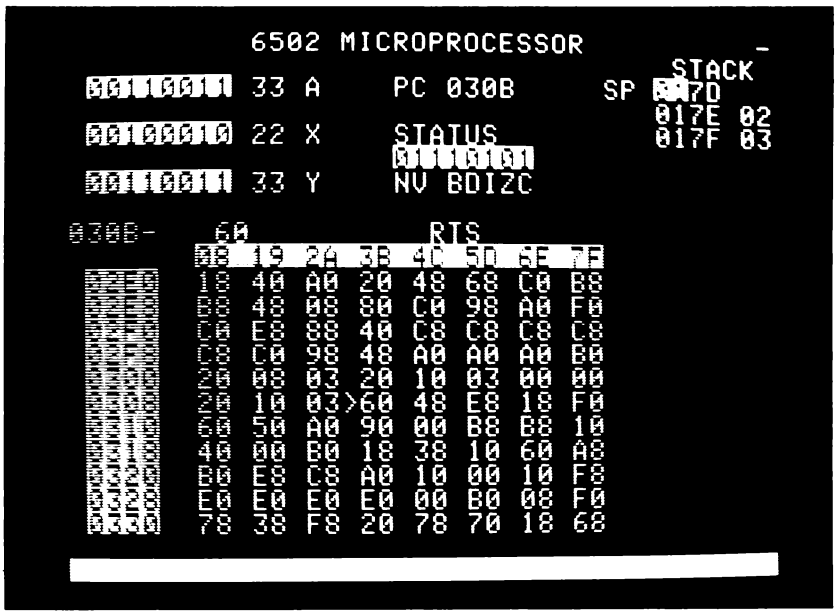


FIGURE 7.6. Screen display after executing RTS at location $310.

If you single step the RTS instruction at location $30B, the program will return to location $303 as shown in Figure 7.7. Note that the program found its way back to location $303 by pulling the last return address (minus 1) from the stack.

The instruction at location $303 is another JSR $310 instruction. Single step this instruction and note how the RTS instruction at $310 will return this time to location $306.

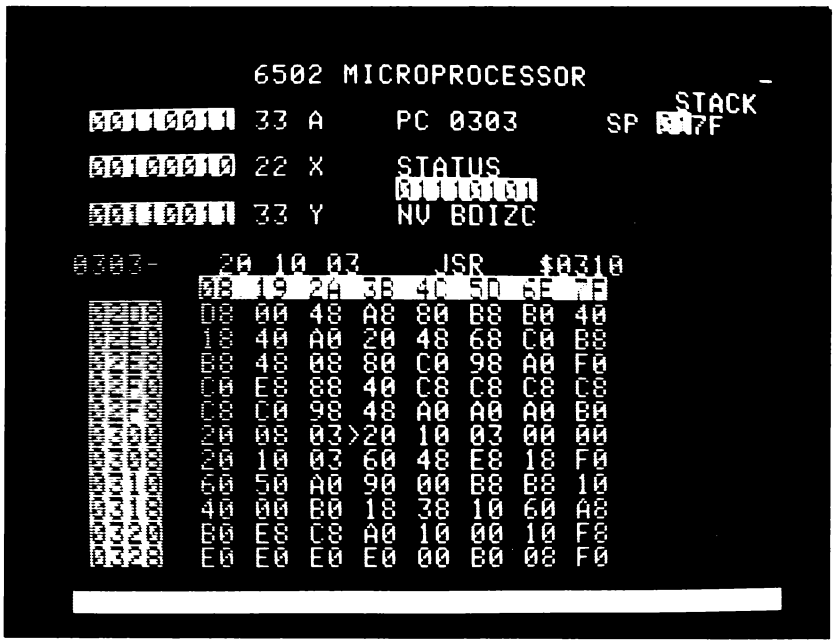


FIGURE 7.7. Screen display after executing RTS at location $30B.

DELAY LOOPS

It is often necessary to include in a program time delays of varying durations. This is done by programming a loop that takes a known length of time to execute. The number of machine cycles used for each instruction is given in the instruction set tables in Appendix A. In the Apple II the duration of each machine cycle is 0.9775 μsec (corresponding to a clock frequency of 1.023 MHz). This information can be used to calculate the total time required to execute various loops.

Delays Less Than 1 msec

Consider the loop shown in Figure 7.8. The first instruction sets the value in the X index register to 0. The next instruction (DEX) decrements this value to $FF. The instruction BNE loop will branch back to the DEX in-

struction until the value in X has been decremented to 0. This means that X will be decremented 256 times. The instruction DEX takes two machine cycles and BNE takes three cycles. Therefore, the total number of cycles required to complete the loop is (2 + 3) × 256 = 1,280. Adding the two cycles for the LDX instruction, the loop takes 1,282 × 0.9775 μsec or 1.253 msec.

```
                                          No. of machine cycles
0300  A2 00          LDX #$00                        2
0302  CA      LOOP    DEX                  2 }
0303  DO FD           BNE LOOP             3 } 5×256 = 1,280
                                                     1,282

      1,282 × 0.9775 μsec = 1,253 μsec
                          = 1.253 msec
```

FIGURE 7.8. A 1.25–msec delay.

Delays of less than 1.25 msec can be achieved by loading X with a value other than 0 in the first instruction. Since the loop takes five cycles to execute, delays can be obtained in increments of about five μsec. For example, suppose that a delay of 200 μsec is needed. From Figure 7.9 we can find the *value* to store in X as follows:

$$(5 \times \text{value} + 2) \times 0.9775 = 200$$
$$5 \times \text{value} + 2 = 200/0.9775 = 205$$
$$5 \times \text{value} = 203$$
$$\text{value} = 40.6$$

```
          LDX #value                2
LOOP      DEX              2 }    5 × value
          BNE LOOP         3 }
                                  5 × value + 2
```

$$(5 \times \text{value} + 2) \times 0.9775 = \text{delay } (\mu\text{sec})$$

FIGURE 7.9. Different values in X will produce different delays.

Loading X with 41 = $29 will result in a delay of 202 μsec. If you need a delay closer to 200 μsec you can add four NOP instructions, which will add 8 cycles. The value to be stored in X can then be calculated from the equation

$$5 \times \text{value} + 10 = 205$$
$$5 \times \text{value} = 195$$
$$\text{value} = 39$$

Therefore, by storing a value of 39_{10} = $27 in X and adding four NOP instructions after the loop in Figure 7.9, a delay of 200 μsec can be achieved.

Delays That Are Multiples of 1 msec

We will write a subroutine that will produce a delay of X msec, where X is the value in the index register X when the subroutine is called. The subroutine will use the accumulator A and the index register X. In order not to change the values in A and X, the subroutine will first push these values onto the stack and then pull them off the stack before returning to the calling program. The subroutine is shown in Figure 7.10.

The first three instructions in the subroutine DELAY push A and X on the stack. These values are pulled off the stack at the end of the subroutine. The number of NOPs used at the beginning of LOOPX and the value of $CA loaded into accumulator A were determined as follows. If N is the number of NOPs and AVAL the value loaded into A, then the number of cycles used each time through the loop LOOPX is

$$\#\ \text{cycles} = 9 + 2N + 5\ \text{AVAL}$$

One time through this loop will equal 1 msec if

$$\begin{aligned}(9 + 2N + 5\ \text{AVAL}) \times 0.9775 &= 1{,}000\\ 9 + 2N + 5\ \text{AVAL} &= 1{,}023\\ 2N + 5\ \text{AVAL} &= 1{,}014\end{aligned}$$

Picking N = 2 will make 5 AVAL be a multiple of 5. Thus

$$\begin{aligned}5\ \text{AVAL} &= 1{,}014 - 4 = 1{,}010\\ \text{AVAL} &= 1{,}010/5 = 202 = \$CA\end{aligned}$$

FIGURE 7.10. Subroutine to produce a delay of X msec.

		JSR DELAY		6	
48	DELAY	PHA		3	
8A		TXA		2	
48		PHA		3	
EA	LOOPX	NOP		2	(13 + 5 × 202)X
EA		NOP		2	
38		SEC		2	
A9 CA		LDA	#$CA	2	
E9 01	LOOPA	SBC	#$01	2	5×202
D0 FC		BNE	LOOPA	3	
CA		DEX		2	
D0 F4		BNE	LOOPX	3	
68		PLA		4	
AA		TAX		2	
68		PLA		4	
60		RTS		6	
				30 + 1,023 X	

Delay = (30 + 1,023 X) × 0.9775 = X msec + 29.3 μsec.

The total number of cycles used when calling this subroutine is 30 + 1,023 X including the 6 cycles used by the JSR instruction. The longest delay this subroutine can produce is about ¼ second. Note that the maximum delay actually occurs for a value of X = 0. The minimum delay of about 1 msec occurs for X = 1. Delays longer than ¼ second can be obtained by calling this subroutine in another loop.

EXERCISE 7.1

Write a subroutine that will produce a 5-second delay.

EXERCISE 7.2

The following delay subroutine is built into the Apple II monitor at location $FCA8. Show that the number of machine cycles used by this subroutine is

$$8 + 12A + 5A^2$$

where A is the value stored in accumulator A when the subroutine is called.

```
WAIT    SEC
WAIT2   PHA
WAIT3   SBC     #$01
        BNE     WAIT3
        PLA
        SBC     #$01
        BNE     WAIT2
        RTS
```

THE APPLE II SPEAKER

The Apple II has a built-in speaker that will allow you to make simple sound effects. Sound is produced by the speaker when a diaphragm made in the form of a paper cone is moved back and forth rapidly. The faster the diaphragm vibrates the higher the frequency of the resulting sound waves. High notes have higher frequencies than low notes.

The speaker on the Apple II is controlled by memory location $C030. Each time this address is put on the address bus, the speaker will toggle. That is, if the diaphragm is out, it will move in; if it is in, it will move out. Thus, if memory location $C030 is referred to over and over again rapidly, the speaker diaphragm should move in and out rapidly and produce a sound.

The assembly language subroutine shown in Figure 7.11 will produce a "square wave" tone of duration bL and half-period aP as shown in Figure 7.12. A hex value of P between 00 and FF is stored in location

```
                    ;           MUSIC TONE
                    ;
                    P           EQU 300
                    L           EQU 301
                                ORG 302
0302-   A0 00       NOTE        LDY #$00
0304-   AE 00 03    N1          LDX P           ;X = pitch
0307-   AD 30 CO                LDA $CO30       ;Toggle speaker
030A-   88          N2          DEY             ;Count to 256
030B-   D0 05                   BNE N3
030D-   CE 01 03                DEC L           ;If tone done
0310-   F0 05                   BEQ N4          ;then exit
0312-   CA          N3          DEX             ;else count pitch time
0313-   D0 F5                   BNE N2
0315-   F0 ED                   BEQ N1          ;End of half-cycle
0317-   60          N4          RTS
```

FIGURE 7.11. Assembly language listing of music tone subroutine.

$300.* A hex value of L is stored in location $301. The subroutine begins at address $302.

Index register Y counts continually from FF to 00 at address $30A. Index register X counts down from P to 0 at address $312. When X decrements to 0, a half-cycle is complete and the program branches back to N1, where P is reloaded with the pitch value and the speaker is toggled again. After Y is decremented 256 times, the value of L (location $301) is decremented once. When L goes to 0 the subroutine is exited and the tone stops.

When Y is nonzero, the loop from $30A to $313 takes 10 machine cycles. For the 1.023-MHz clock of the Apple II, the period of a half-cycle of the sound wave will then be $9.775 \times P$ μsec. The period of a full cycle will be $19.55 \times P$ μsec and the frequency of the sound wave will then be

$$\text{freq.} = \frac{1}{\text{period}}$$

$$= \frac{1}{19.55 \times P \ \mu\text{sec}}$$

$$= \frac{10^6}{19.55 \times P}$$

from which

$$P = \frac{51{,}151}{\text{freq.}}$$

*The assembler directives EQU and ORG in Figure 7.11 will be defined in Chapter 8.

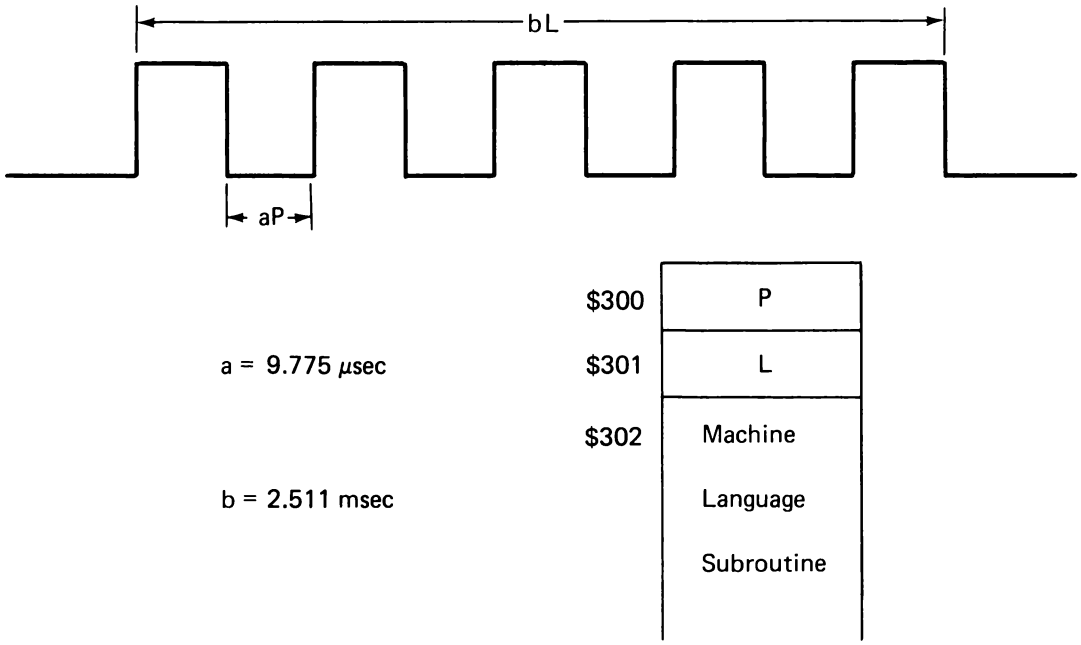


FIGURE 7.12. The subroutine in Figure 7.11 will produce a tone of pitch P and duration L.

Every time that Y decrements to 0 the statements at address $30D and $310 take 9 cycles to execute. Neglecting the extra 11 cycles that occur every time the speaker toggles, the total time required for Y to decrement 256 times is

$$256 \times 9.775\ \mu\text{sec} + 9 \times 0.9775\ \mu\text{sec} = 2.511\ \text{msec}$$

The duration of the tone will therefore be approximately $2.511 \times L$ msec.

Setting Breakpoints

In order to test the speaker routine in Figure 7.11, type it in starting at location $302. Then type in the following statements, starting at location $320.

```
0320   20   02   03    JSR   NOTE
0323   20   02   03    JSR   NOTE
0326   20   02   03    JSR   NOTE
0329   00              BRK
```

Now go to memory location $323 and press /B. The message

BREAKPOINT: S N F C

will appear on the command line as shown in Figure 7.13. Now press S. This will set a breakpoint at location $323 by changing the op-code 20 to 00. If the program starting at location $320 is now executed, it will stop when it gets to the breakpoint you set at $323.

Before executing the program at $302 you must store some pitch value P in $300 and a duration value L in $301. Go to $300 and store the value $80 in $300 and $FF in $301. You are now ready to execute the program at $302.

Executing Programs

You can execute a machine language program by pressing /E. When you do this, the command line will read

EXECUTE: A G R

Press A. You can now enter the starting address of the program (320), as shown in Figure 7.14. When you press RETURN after entering the starting address, the program at $320 will be executed. You should hear a tone that lasts for about half a second. The program will stop at location $323, where you set the breakpoint. Note that the original op-code (20) has automatically replaced the breakpoint op-code (00). Also note that the duration $FF in location $301 has been decremented to 0 (by the instruction in $30D). You can leave it at 0 and it will decrement one more time than FF. You can resume execution at this point by typing /ER. Try it. The subroutine should be called two more times, stopping at the BRK instruction at $329.

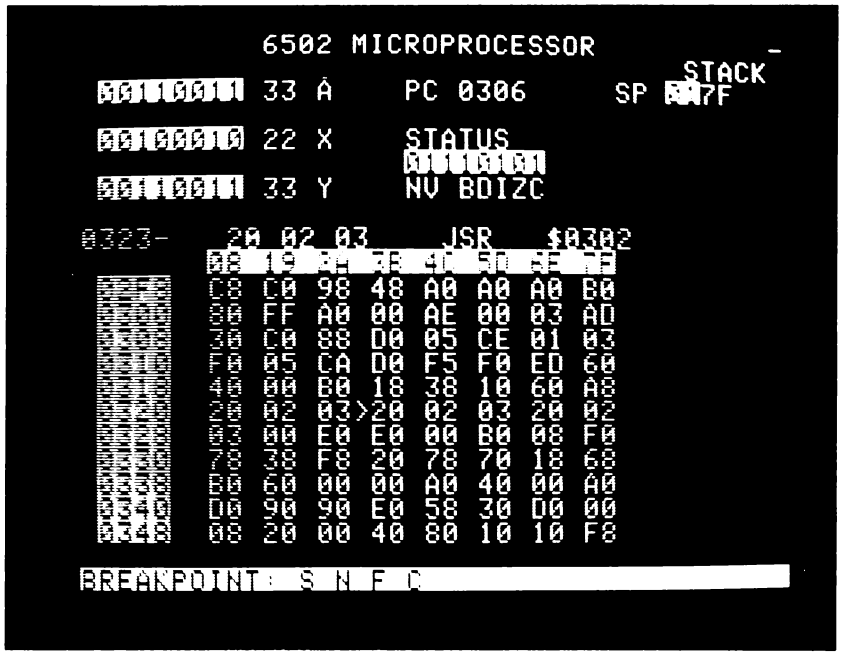
Go back to location $320 and press /EG. Pressing G (go) after /E will begin execution starting at the current location of the position cursor.

Now go to location $326 and set a breakpoint by typing /BN. The N means *no automatic update* of the op-code. Execute the program again starting at $320. Note that when the breakpoint is reached, the original op-code has not replaced the breakpoint. This will allow you to execute the program more than once without having to reset the breakpoint.

In order to clear the breakpoint and replace it with the original op-code, type /BC. If you have set a breakpoint and want to find it, type /BF.

THE APPLE II KEYBOARD

The two memory locations $C000 and $C010 are special locations that are used by the Apple II keyboard. As long as no key is pressed, bit 7 (the sign bit) of $C000 will be 0. Thus, you can see if any key has been



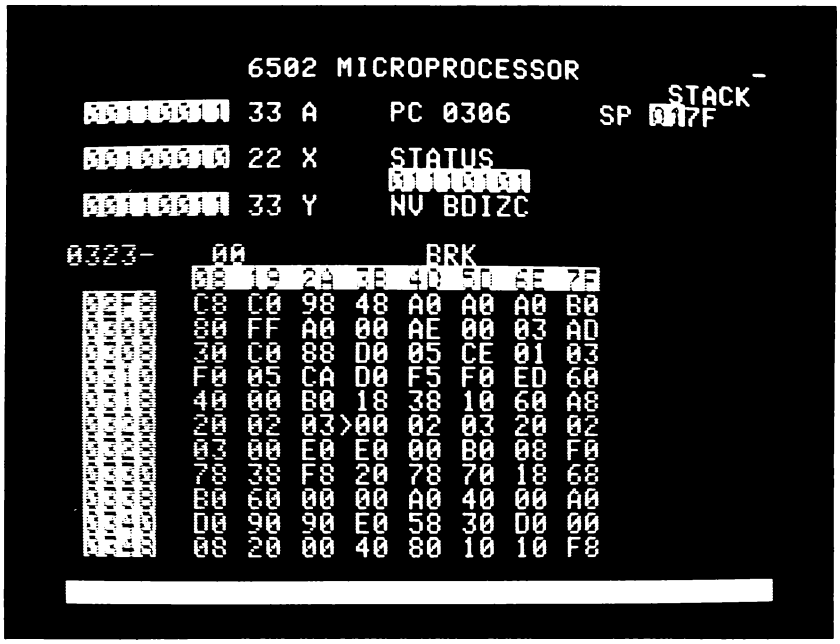


FIGURE 7.13. /BS will set a breakpoint (op-code 00) at the location of the position cursor.

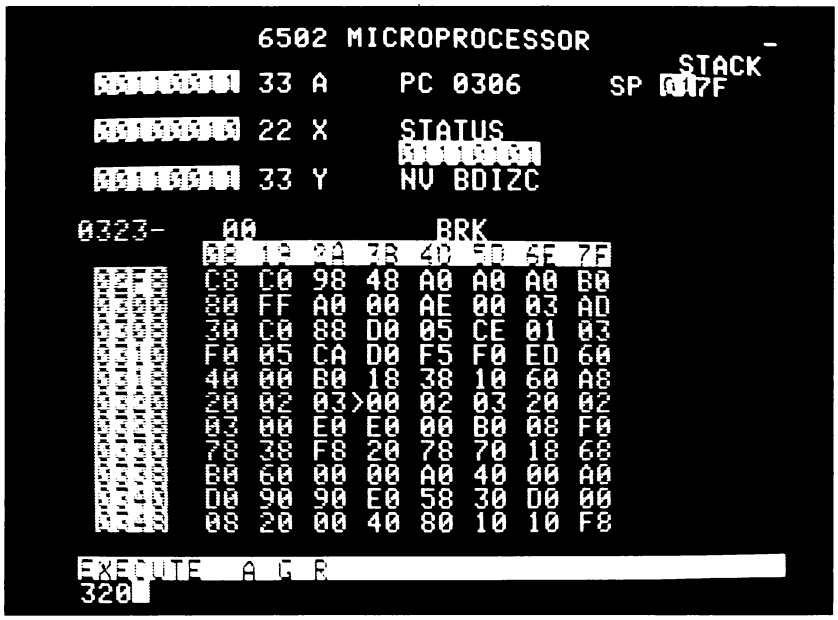


FIGURE 7.14. /EA will allow you to enter a starting address and then execute the program starting at that address.

pressed by reading memory location $C000 and then checking the negative flag N in the status register. For example, the loop

```
KEYIN LDA $C000
      BPL KEYIN
```

will continue looping until a key is pressed.

If a key has been pressed, the value stored in memory location $C000 will be equal to the ASCII code for that key. This value will, of course, have bit 7 set to 1 (otherwise, a key would not have been pressed). But if this is the case, how can you tell when you press a second key? You must somehow reset bit 7 in location $C000, to 0 so that it can be set to 1 again when you press a second key. You accomplish this reset function by referring to memory location $C010. It doesn't matter how you refer to it—you can use either a load statement such as

```
LDA $C010
```

or a store statement such as

```
STA $C010
```

or the BIT statement

```
BIT $C010
```

Simply referring to location $C010 will cause bit 7 in location $C000 to be reset to 0.

The ASCII codes for all of the keys on the Apple II keyboard are shown in Figure 7.15. Note that bit 7 is set to 1 for all of these codes. You should compare these codes with the standard ASCII codes and the Apple II ASCII codes given in Figures 3.5 and 3.6 in Chapter 3.

Key	Alone	CTRL	SHIFT	Both	Key	Alone	CTRL	SHIFT	Both
space	$A0	$A0	$A0	$A0	RETURN	$8D	$8D	$8D	$8D
0	$B0	$B0	$B0	$B0	G	$C7	$87	$C7	$87
1!	$B1	$B1	$A1	$A1	H	$C8	$88	$C8	$88
2"	$B2	$B2	$A2	$A2	I	$C9	$89	$C9	$89
3#	$B3	$B3	$A3	$A3	J	$CA	$8A	$CA	$8A
4$	$B4	$B4	$A4	$A4	K	$CB	$8B	$CB	$8B
5%	$B5	$B5	$A5	$A5	L	$CC	$8C	$CC	$8C
6&	$B6	$B6	$A6	$A6	M	$CD	$8D	$DD	$9D
7'	$B7	$B7	$A7	$A7	N^	$CE	$8E	$DE	$9E
8(	$B8	$B8	$A8	$A8	O	$CF	$8F	$CF	$8F
9)	$B9	$B9	$A9	$A9	P@	$D0	$90	$C0	$80
:*	$BA	$BA	$AA	$AA	Q	$D1	$91	$D1	$91
;+	$BB	$BB	$AB	$AB	R	$D2	$92	$D2	$92
,<	$AC	$AC	$BC	$BC	S	$D3	$93	$D3	$93
-=	$AD	$AD	$BD	$BD	T	$D4	$94	$D4	$94
.>	$AE	$AE	$BE	$BE	U	$D5	$95	$D5	$95
/?	$AF	$AF	$BF	$BF	V	$D6	$96	$D6	$96
A	$C1	$81	$C1	$81	W	$D7	$97	$D7	$97
B	$C2	$82	$C2	$82	X	$D8	$98	$D8	$98
C	$C3	$83	$C3	$83	Y	$D9	$99	$D9	$99
D	$C4	$84	$C4	$84	Z	$DA	$9A	$DA	$9A
E	$C5	$85	$C5	$85	←	$88	$88	$88	$88
F	$C6	$86	$C6	$86	→	$95	$95	$95	$95
					ESC	$9B	$9B	$9B	$9B

FIGURE 7.15. ASCII codes associated with the Apple II keyboard.

The program shown in Figure 7.16 will wait for you to press a key, and will then use the ASCII code for that key as the pitch value P in the tone subroutine of Figure 7.11 (assuming you have already typed in the subroutine in Figure 7.11). Type in this program starting at location $320. Make sure you have also typed in the subroutine in Figure 7.11.

```
0320   A9 00          LDA #$00
0322   8D 01 03       STA $301      ;max duration
0325   AD 00 C0 KEYIN LDA $C000     ;loop until
0328   10 FB          BPL KEYIN     ;key pressed
032A   2C 10 C0       BIT $C010     ;clear bit 7
032D   8D 00 03       STA $300      ;P = ASCII code
0330   20 02 03       JSR $302      ;play tone
0333   4C 25 03       JMP KEYIN     ;do again
```

FIGURE 7.16. Program to play a tone whose pitch value corresponds to the ASCII code of the key pressed.

Execute the program starting at location $320. Press different keys in order to "hear" the ASCII codes. Note that the larger the ASCII code, the lower the frequency of the tone.

The second statement in the program in Figure 7.16 is

```
8D 01 03      STA $301
```

which means "Store the contents of accumulator A in memory location $301." The statement at location $32A is

```
2C 10 C0      BIT $C010
```

which means "AND the contents of accumulator A with the contents of memory location $C010." The purpose of this statement is just to reset bit 7 of $C000. Note that the address $C010 is written low byte first in the machine language code 2C 10 C0. This is always the case in 6502 machine language code. For example, the last statement in Figure 7.16 is

```
4C 25 03      JMP KEYIN
```

which jumps to location $325. These are examples of the absolute addressing mode in which the absolute address of a memory location is specified in the low-byte, high-byte format. Other addressing modes that are available with the 6502 will be described in the next chapter.

EXERCISE 7.3

Modify the program in Figure 7.16 so that each note is played one octave higher. Hint: Doubling the frequency will produce a tone one octave higher. The frequency will be doubled if the pitch value P stored in memory location $300 is divided by 2. An 8-bit byte can be divided by 2 by shifting right one bit (logic shift right, LSR).

8

Addressing Modes

In previous chapters we have executed a number of 6502 instructions, either by single stepping (using CTRL S) or by executing a program using /E. Each 6502 op-code has associated with it an addressing mode which determines on what data the operation is to be performed. Some instructions, such as PHA, have only a single addressing mode. Others, such as LDA, can have as many as eight different addressing modes. In the case of the instruction LDA, the addressing mode determines where the byte of data is located that is to be loaded into accumulator A. The tables in Appendix A give the op-codes associated with the different addressing modes for all of the 6502 instructions.

In this chapter you will learn to use the following 6502 addressing modes:

1. inherent or accumulator
2. immediate
3. zero page or direct
4. absolute or extended
5. relative
6. indexed, including zero page and absolute indexed
7. indirect, including preindexed and postindexed indirect

The listing given in Figure 7.11 is an example of an assembly language subroutine. When writing an assembly language program you first write the program using the instruction mnemonics. Such instructions may contain up to four fields in the following order:

```
LABEL   INSTRUCTION-MNEMONIC   OPERAND   COMMENT
```

For example, in the statement

```
N1   LDX   P        ;X = pitch
```

the label is N1, the instruction mnemonic is LDX, the operand is P, and the comment is ;X = pitch (see Figure 7.11). Only the instruction mnemonic is required in all assembly language statements, as illustrated in Figure 7.11. The form of the operand indicates which addressing mode is being used. Thus, the combination of the instruction mnemonic and the operand will determine which op-code in the tables in Appendix A should be used.

When using one of the *assemblers* that are available for the Apple II, you would type in the assembly language program, starting with the label field, in the form shown in Figure 7.11. The assembler would then generate the machine language code shown at the left in Figure 7.11. The assembler would look up all of the op-codes for you and figure out the proper byte to use for the operands and branching offsets.

You must give an assembler certain additional information. You do this by means of *assembler directives*. For example, in Figure 7.11 the statement

```
P   EQU    300
```

is an *equate* directive that equates the symbol P to the hex value $300. The statement LDX P will then load the index register X with the value stored in memory location $300.

The ORG assembler directive tells the assembler the starting address of the machine language or object code of the program. Thus, in Figure 7.11, the statement

```
ORG   302
```

means that the op-code of the first instruction (A0, LDY) will be stored in memory location $302.

In this chapter you will learn how to do hand assembly. That is, you will learn how to look up the proper op-codes for the various addressing

modes, using the tables in Appendix A. We will consider each addressing mode separately.

INHERENT OR ACCUMULATOR ADDRESSING MODE

There are several 1-byte instructions for which the location of the data to be operated on is inherent in the op-code itself. You have already used most of these instructions. For example, the LSR instruction 4A operates on the contents of accumulator A (see Figure 4.6). The instructions INX (E8) and DEX (CA) operate on the contents of the index register X (see Table 4.2). The op-codes associated with the inherent/accumulator addressing mode are given in the first column of Table A-1 in Appendix A. Note that these are all 1-byte instructions. (The number of bytes is given in the column under the # sign). The number following the op-code in Table A-1 (under the ~ sign) is the number of machine cycles used in executing the instruction.

IMMEDIATE ADDRESSING MODE

You have already used the immediate mode of addressing several times. For example, in Figure 5.1 the instruction

```
A9 35      LDA   #$35
```

loads accumulator A with the value $35. The symbol # is used to tell an assembler that the immediate mode of addressing is being used. In the immediate addressing mode the data is found *immediately* following the op-code.

The second addressing mode column in Table A-1 gives the op-codes for the immediate mode. The op-code in this column opposite LDA is A9. This is a 2-byte instruction in which the second byte consists of the data to be loaded into accumulator A. Thus, the 2 bytes A9 35 will load the value $35 into accumulator A.

Study Table A-1 and note the instructions that have an immediate addressing mode. You have already used ADC and SBC in the immediate mode in Chaper 5 (see Figures 5.1 and 5.11). You also used LDY and CPY in the immediate mode in Figure 6.5. Study these examples again and note how the op-codes can be found from Table A-1.

We will look at three more examples involving the logical instructions AND, ORA, and EOR. These logical instructions perform the logical operations AND, OR, and EOR, defined on a bit-by-bit basis according to Table 8.1.

Table 8.1 Logical Operations

b_A	b_M	b_A AND b_M	b_A OR b_M	b_A EOR b_M
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

The instruction AND performs a bit-by-bit AND operation of the contents of accumulator A with the contents of memory location M, where M is specified by the addressing mode. For the immediate mode, M will be the byte following the op-code. Type in the two instructions in Figure 8.1 and execute them in the single-step mode. Note that after executing the instruction AND #$F0, the value in accumulator A will be $50. That is, the lower nibble (A) has been masked to 0. This is because ANDing any bit with a 0 produces a 0, while ANDing any bit with a 1 leaves the bit unchanged (See Table 8.1).

```
0300   A9 5A     LDA  #$5A
0302   29 F0     AND  #$F0
```

FIGURE 8.1. ANDing a byte with $F0 will mask the lower nibble.

The instruction ORA performs a bit-by-bit OR operation on the contents of accumulator A with the contents of memory location M. A particular bit can be set to 1 by using the ORA instruction. For example, the instructions in Figure 8.2 will set bit 7 of accumulator A by ORing the contents of A ($13) with $80. Type in these two instructions and execute them by single stepping.

```
0300   A9 13     LDA  #$13
0302   09 80     ORA  #$80
```

FIGURE 8.2. Bit 7 of accumulator A can be set by executing ORA #$80.

The instruction EOR performs a bit-by-bit exclusive-OR operation on the contents of accumulator A with the contents of memory location M.Note from Table 8.1 that

$$0 \text{ EOR } 1 = 1$$

and

$$1 \text{ EOR } 1 = 0$$

Thus, if you exclusive-OR the contents of A with $FF, you will obtain the one's complement of A. For example, type in the two instructions in Figure 8.3 and execute them by single stepping. Note that the value $55 becomes $AA when exclusive-ORed with $FF.

```
0300    A9 55    LDA #$55
0302    49 FF    EOR #$FF
```

FIGURE 8.3. The one's complement of A can be found by executing EOR #$FF.

ZERO PAGE OR DIRECT ADDRESSING MODE

The addressing mode given in the third column of Table A-1 is the zero page (or direct) addressing mode. Note that all instructions using this addressing mode are 2-byte instructions. The first byte is the op-code and the second byte is the *address* of the data. Since this address is only a single byte it can have only values $00–$FF. This value is taken to be the low-order byte of the address. The high-order byte is assumed to be $00. That is, the data referred to are in the zero page of memory. Thus, zero page addressing is used to access data that are stored between addresses $0000 and $00FF.

Zero page addressing is used to save memory because the instructions take only 2 bytes. As we will see, absolute addressing takes 3 bytes because the operand address uses 2 bytes instead of one. For this reason, data are normally stored on page 0 in 6502 programs to take advantage of zero page addressing.

An example of zero page addressing is given in Figure 8.4. Type in this program and single step through it. The first instruction clears the carry flag. Note from the tables in Appendix A that none of the other instructions in this program (including INC) changes the carry flag. Thus, the carry will remain cleared and the BCC instruction will always branch to LOOP.

```
0098    18          CLC
0099    A9 00       LDA #$00
009B    85 90       STA $90
009D    E6 90  LOOP INC #90
009F    90 FC       BCC LOOP
```

FIGURE 8.4. Example of zero page addressing.

The instruction STA $90 uses the zero page addressing mode and will store the value in accumulator A ($00) in memory location $90. You should see this happen by observing the contents of $0090 as you single step through this program.

The instruction INC $90 also uses the zero page addressing mode. Each time this instruction is executed the value in memory location $0090 is incremented by 1. Press the RESET key to exit this program.

EXERCISE 8.1

Modify the program in Figure 8.4 so that the value in memory location $0090 is *decremented* by 1 each time through the loop.

ABSOLUTE OR EXTENDED ADDRESSING MODE

The absolute, or extended, addressing mode uses a 2-byte operand that can represent any address in the 6502 memory map. The third instruction in Figure 7.11 uses the absolute addressing mode:

```
AD 30 C0        LDA $C030
```

Note that this is a 3-byte instruction in which the operand address is stored in the program in the order low-byte, high-byte. In Table A-1 the op-code AD is in the absolute addressing mode column opposite the LDA instruction.

We will further illustrate the absolute addressing mode using the BIT instruction. As seen from Table A-1 this instruction uses only the zero page and absolute addressing modes. The BIT instruction is normally used to test a particular bit in a memory location. It can do this in three different ways. All three ways are illustrated in the program shown in Figure 8.5. Type in this program and single step through it.

```
0308    A9 C5     LOOP    LDA #$C5
030A    8D 00 03          STA $300
030D    A9 04             LDA #$04
030F    2C 00 03          BIT $300
0312    30 F4             BMI LOOP
```

FIGURE 8.5. Using the BIT instruction with absolute addressing.

When the statement BIT $300 is executed, the following three things occur:

1. The bits in memory location $300 are ANDed with the corresponding bits in accumulator A and the zero flag, Z, is set accordingly.
2. The value of bit 7 in memory location $300 is assigned to the negative flag, N.
3. The value of bit 6 in memory location $300 is assigned to the overflow flag, V.

Any bit in $300 can be tested by loading accumulator A with only the bit position to be tested set. For example, a value of $04 in A will test bit 2, as shown in Figure 8.6. If bit 2 in $300 is set, the result of the BIT operation will be nonzero and the Z flag will be equal to 0. You can then use the branching instructions BEQ or BNE.

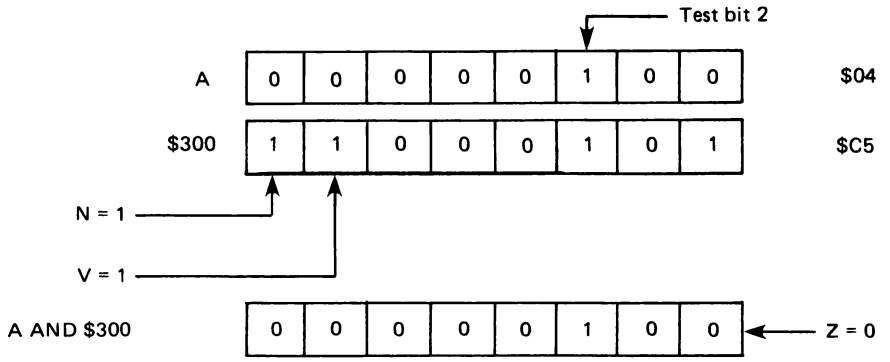


FIGURE 8.6. BIT $300 will test bit 7, bit 6, or any bit set in A.

It is even easier to test bit 6 or bit 7 of $300. You don't have to load any particular value into accumulator A to test these bits. Since the value of bit 7 in location $300 is assigned to the N flag, the statement BIT $300 can be used to test the sign of the value in $300. You can then follow the statement BIT $300 with either BMI or BPL.

Similarly, you can test bit 6 in $300 by following the statement BIT $300 with either BVC or BVS.

RELATIVE ADDRESSING MODE

The relative addressing mode is used only for branching instructions, as described in Chapter 6. All branching instructions are 2-byte instructions in which the second byte is the two's complement offset used to calculate the destination address relative to the program counter plus 2. Note from the tables in Appendix A that the branching instructions take three machine cycles to execute if the branch does not cross a page boundary. If the branch does cross a page boundary, four machine cycles are required to execute a branching instruction.

Remember that the branching offset must be in the range -128_{10} to $+127_{10}$ bytes. Normally, in well-written programs, this does not pose a problem. If on occasion you do need a larger offset you can combine a branching instruction with a JMP instruction that uses the absolute ad-

dressing mode. For example, suppose that you want to execute the statements

```
        BCC LOOP
NEXT    ------
```

where the label LOOP is more than 128 bytes from NEXT. You can replace this with

```
        BCS NEXT
        JMP LOOP

NEXT    ------
```

Note that the statement at NEXT is still executed if the carry flag is set and the statement at LOOP is executed if the carry flag is 0. However, the JMP statement uses the absolute addressing mode so that LOOP can be anywhere in memory.

INDEXED ADDRESSING MODES

The indexed addressing modes use the value in an index register (X or Y) to help calculate the effective address of the data. In all cases the value in the index register is added to the operand address to obtain the effective address. The operand address can be either a zero page address or an absolute address.

Zero Page Indexed Addressing

From Table A-1 you can see that the zero page indexed addressing mode uses the X index register for all instructions except LDX and STX, which use the Y register. Instructions using zero page indexed addressing take 2 bytes. The op-code is followed by a zero page address. The value in the X index register is added to this zero page address to form the effective address, which must also be on page 0. This means that wraparound occurs if the sum exceeds $FF. For example, suppose that the index register X contains the value $12 and the following instruction is executed:

```
B5 F3        LDA $F3, X
```

The effective address is calculated as follows:

```
    $F3
+ $12
ignore carry 1 05
```

Therefore, accumulator A will be loaded with the value in memory location $0005.

Store the 4 bytes 11 22 33 44 in memory locations $0080–$0083. Suppose that you want to move these 4 bytes from $0080–$0083 to $0088–$008B. The program shown in Figure 8.7, which starts at location $0090, will do this. This program first moves the byte in $0083 to $08B and then works backward in memory until the value in $0080 is moved to $0088. Type in this program and single step through it, observing the

```
0090    A2 03          LDX #$03
0092    B5 80    LOOP  LDA $80, X
0094    95 88          STA $88, X
0096    CA             DEX
0097    10 F9          BPL LOOP
```

FIGURE 8.7. The 4 bytes in $0080–$0083 are transferred to $0088–$008B.

transfer of the 4 bytes. After four times through the loop the value of X will be $FF, which sets the negative flag N. Therefore, the BPL test will fail and the loop will be exited.

Absolute Indexed Addressing

Instructions using absolute indexed addressing have 2-byte operand addresses. The value in index register X or Y is added to the operand address to form the effective address. As an example, the program in Figure 8.8 will clear the 256 bytes of memory from $2880 through $297F. Type in this program and execute it, starting at location $300. Examine the memory locations between $2880 and $297F.

```
0300    A9 00          LDA #$00
0302    A0 00          LDY #$00
0304    99 80 28 LOOP  STA $2880,Y
0307    C8             INY
0308    DO FA          BNE LOOP
030A    00             BRK
```

FIGURE 8.8. Program to clear 256 bytes starting at location $2880.

EXERCISE 8.2

After executing the program in Figure 8.8, write and execute a program that will transfer 256 bytes starting at location $2880 to another location starting at address $3880.

INDIRECT ADDRESSING MODES

The instruction

```
4C 10 03        JMP $0310
```

will cause the program to jump to memory location $310. This is an example of absolute addressing. On the other hand, the instruction

```
6C 18 03        JMP ($318)
```

will cause the program to jump to the *address that is stored* in locations $318 and $319. This is an example of absolute *indirect addressing.*

To test this addressing mode, store the value $00 in location $318 and $03 in location $319. Then type in the instructions shown in Figure 8.9. When you single step the instruction at $300, the program will jump to address $310. When you single step the instruction at $310, the program will jump back to location $300. This will occur because the program first goes to location $318, where it finds the address $300 (00 03), to which it really jumps.

```
0300    4C 10 03    JMP $0310
0310    6C 18 03    JMP ($0318)
0318    00 03
```

FIGURE 8.9. Example of indirect addressing.

The JMP instruction is the only 6502 instruction that has absolute indirect addressing. There are many instructions, however, that have a more powerful type of *indexed indirect addressing.* There are two different types of indexed, indirect addressing: preindexed indirect using the X index register, and postindexed indirect using the Y index register.

Preindexed, Indirect Addressing

Indexed, indirect addressing modes use a 2-byte address that must be stored on page 0. The preindexed, indirect addressing mode must use the X index register. Instructions using this mode contain 2 bytes and the operand is written in assembly language as (IND,X). The value of IND is the

second byte of the instruction. This value is added to the value in the X index register to obtain a new zero page address much the same as with zero page indexed addressing. However, in this case the new zero page address points to 2 bytes on page 0 that contain the effective address of the operand. This situation is shown in Figure 8.10.

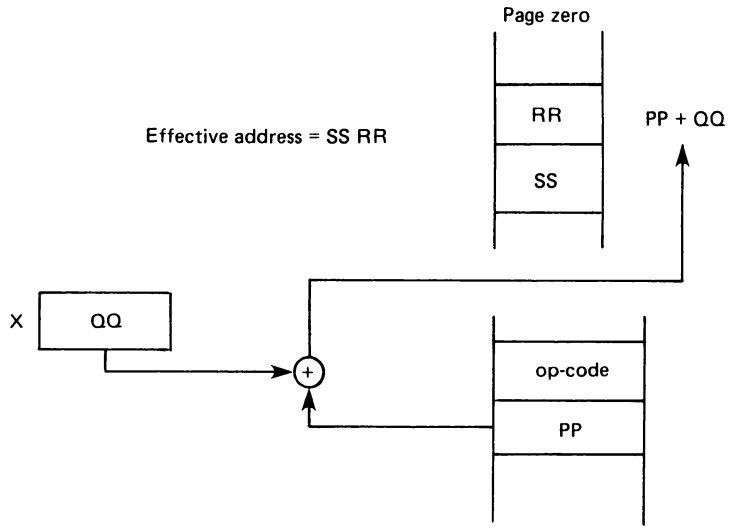


FIGURE 8.10. Preindexed, indirect addressing.

As an example of preindexed, indirect addressing, type in the values shown in Figure 8.11. The address $0300 is stored (low byte first) in locations $0094 and $0095. The instruction at $308 loads the X index regist-

```
0094  00
0095  03

0308  A2  04        LDX  #$04
030A  A9  AA        LDA  #$AA
030C  81  90        STA  ($90,X)
```

FIGURE 8.11. Example of preindexed, indirect addressing.

er with the value $04. The next instruction loads the accumulator A with the value $AA. The instruction STA ($90,X) adds the value in X ($04) to $90 and then goes to this address ($94) to find the effective address. This effective address is $0300. Therefore, the value $AA is stored in location $0300. Single step through these three instructions and note that the value $AA is stored in location $0300.

The preindexed, indirect addressing mode can be used when you have a table of special addresses, possibly associated with different I/O ports. You can access these different addresses by changing the value in the index register X.

This addressing mode is most often used with an X value of 0. In this case the effective address is stored at the address of the operand.

Postindexed, Indirect Addressing

The postindexed, indirect addressing mode must use the Y index register. Instructions using this mode contain 2 bytes and the operand is written in assembly language as (IND),Y. The value of IND is the second byte of the instruction. This byte is a zero page address that points to 2 bytes on page 0. The value in the Y index register is added to this 2-byte address on page 0 to form the effective address. The situation is shown in Figure 8.12.

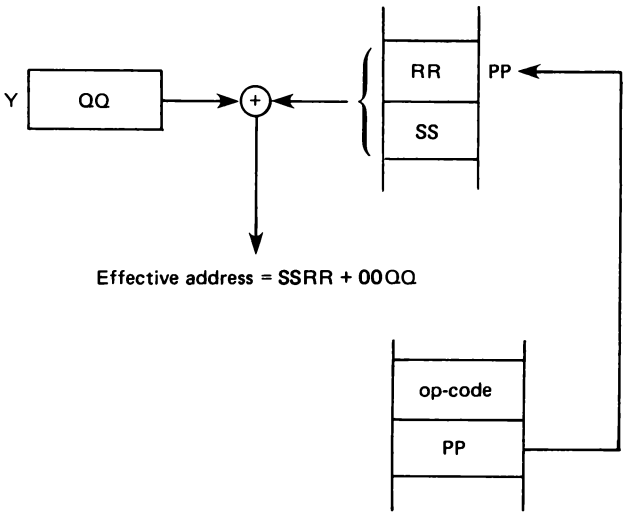


FIGURE 8.12. Postindexed, indirect addressing.

As an example of using postindexed, indirect addressing, type in the values shown in Figure 8.13. The address $0300 is stored (low byte first) in locations $0094 and $0095. The instruction at $308 loads the Y index register with the value $04. The next instruction loads the accumulator A with the value $BB. The instruction STA ($94),Y goes to location $0094 and gets the address $0300. It then adds the value in the Y index register

```
0094  00
0095  03

0308  A0 04       LDY  #$04
030A  A9 BB       LDA  #$BB
030C  91 94       STA  ($94),Y
```

FIGURE 8.13. Example of postindexed, indirect addressing.

($04) to this address to form the effective address $0304. Therefore, the value $BB is stored in location $0304. Single step through these instructions and note that the value $BB is stored in location $0304. Compare this example with that shown in Figure 8.11 and note carefully the difference between preindexed and postindexed indirect addressing.

The example shown in Figure 8.8 cleared 256 bytes starting at location $2880. In order to clear more bytes, or to start at a different memory location, we need to make the address $2880 variable. This is what postindexed, indirect addressing allows us to do. The example shown in Figure 8.14 will clear 8K bytes of memory starting at location $2000. That is, it will assign a value of $00 to all bytes between $2000 and $3FFF.

Type in this program, making sure to store the address $2000 in memory locations $0090 and $0091. Execute the program starting at location $0300 and then examine the memory locations between $2000 and $3FFF.

```
0090  00          starting address  L0
0091  20          starting address  H1

0300  A0  00            LDY  #$00
0302  A9  00    LOOP1   LDA  #$00
0304  91  90    LOOP2   STA  ($90),Y
0306  C8                INY
0307  D0  FB            BNE  LOOP2
0309  E6  91            INC  $91
030B  A5  91            LDA  $91
030D  C9  40            CMP  #$40
030F  D0  F1            BNE  LOOP1
0311  00                BRK
```

FIGURE 8.14. Program to clear 8K bytes from $2000 to $3FFF.

Study the program in Figure 8.14 and make sure that you understand how it works. Note that after each 256-byte block of memory is cleared, the value of Y will be 0 again. The high-order byte of the starting address in $0091 is then incremented and the next 256-byte block of memory is cleared. The process is stopped when the high-order byte of the starting address reaches $40.

EXERCISE 8.3

Modify the program in Figure 8.14 to store the hex value $AA in all memory locations between $2000 and $3FFF. Remember that location $0091 is changed by the program and must be reloaded each time the program is run.

EXERCISE 8.4

Write a program that will move 1K bytes starting at memory location $2000 to a new location starting at address $4000.

9

Displaying Characters on the Screen

In this chapter we begin to look at how the Apple II displays information on the video screen. Chapter 9 will consider the display of text; low-resolution graphics will be covered in Chapter 10; and high-resolution graphics will be described in Chapter 11.

In this chapter you will learn

1. how raster scan displays work
2. how to find the address of any byte in the Apple II's TV RAM
3. the screen ASCII codes used by the Apple II
4. how to print any character on the screen
5. how to print a message on the screen
6. the Apple II built-in routines for printing characters on the screen

RASTER SCAN DISPLAYS

A TV video screen works by causing an electron beam to scan the screen in a raster format—that is, in a series of horizontal lines moving down the screen. The electron beam impinging on a phosphor-coated screen causes light to be emitted; it scans the entire screen in 1/60 second. Home TV images are displayed in an *interlaced* format which displays every other scan line in the first 1/60 second and the remaining alternate scan lines in the next 1/60 second. It therefore takes 1/30 second to display the entire image.

Although only one spot on the screen is being hit by the electron beam at any instant of time, if the entire screen is rewritten every 1/30 second, the eye will retain the image from one screen scan to the next. It will therefore look as if an image fills the entire screen, even though only one spot is really being displayed.

The Apple II scans 192 horizontal lines every 1/60 second in a non-interlaced format. Eight consecutive horizontal lines are used to display a single character. Therefore, 24 (192/8) lines of text can be displayed on the screen. Each character displayed on the screen is in the form of a 5×7-dot matrix located within a 7×8-dot area on the screen, as shown in Figure 9.1.

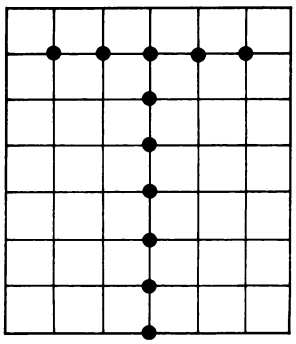


FIGURE 9.1.
Each 5x7-dot matrix character occupies a 7x8-dot area on the screen.

A dot will be displayed on the screen if the electron beam is turned on when it is at the location on the screen where the dot is to be displayed. If the electron beam is turned off, no dot will be displayed on the screen. The video signal that is fed into the TV turns the electron beam on and off in the proper timing to produce the desired effect on the screen.

For example, to display the T shown in Figure 9.1, we must control the electron beam for eight consecutive scan lines. The top line is blank, which means that the electron beam is off the entire time it scans across the seven dot positions. The second line (the top of the T) contains five dots with a blank on each side. If we represent a dot by 1 and a blank by 0, then the top of the T is represented by 0111110. The next six scan lines of the T would each be represented by 0001000, which would display the vertical part of the T.

The Apple II displays 40 characters per line. This will use 280 (40×7) dot positions on each line. To display an entire line of 40 characters we must display the top row of each character, then the second row, then the third row, and so on. It will take eight horizontal scan lines to display one line of 40 characters.

The way that this is done is shown in Figure 9.2. Suppose that you want to display the three characters T H E at the upper-left-hand corner

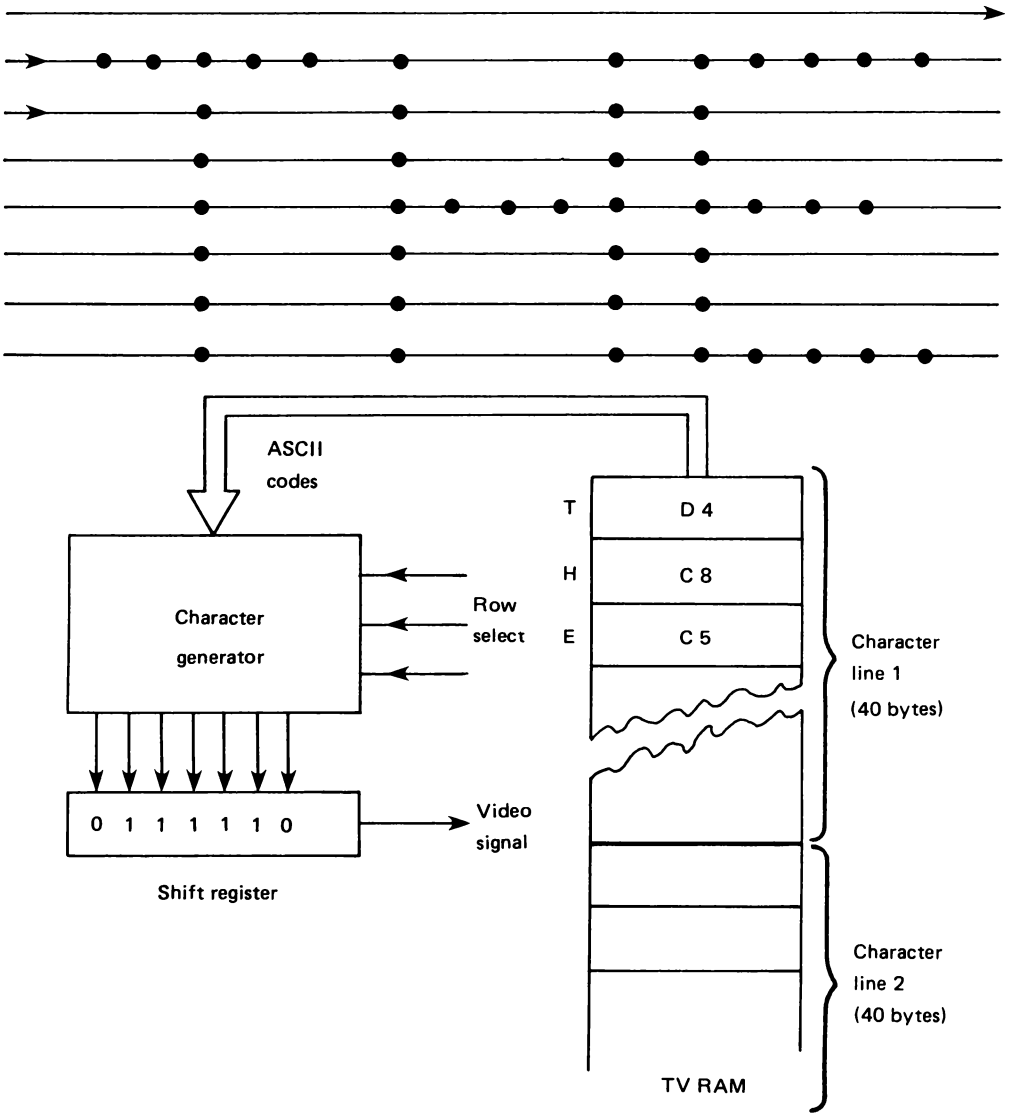


FIGURE 9.2. ASCII codes for characters to be displayed on–are stored in a TV RAM.

of the screen. The screen ASCII codes for these characters are stored in the first three locations of a TV RAM. To display the top bar of the T (in row 1), the ASCII code for T ($D4) and the row number (01) are used to form an input address to the character generator. The character generator is really a read only memory (ROM) that contains the dot patterns used to form the various characters. For example, the top bar of the T is represented by the dot pattern 0111110. This will be the output of the character generator when the input is the ASCII code for T and the row select value 01. This dot pattern code is put into a shift register from which it is

shifted out 1 bit at a time to form the video signal. This video signal goes to the TV where it controls the electron beam.

After the top of the T is displayed, the top of the H must be displayed next. The ASCII code for H (stored in the next location of the TV RAM) is moved to the input of the character generator. The row select input remains at 01. The output of the character generator will be 0100010, corresponding to the top part of the H. This code then is shifted out of the shift register and becomes the next part of the video signal.

This process continues until the first 40 ASCII codes, stored in the first 40 bytes of the TV RAM, have all been cycled to the character generator. At this point the top of all of the characters will have been displayed. This will all have taken place in less than one ten-thousandth of a second. The row select input is then incremented by 1 and the same 40 ASCII codes stored in the first 40 bytes of the TV RAM are recycled to the character generator. This will cause the proper dot pattern for row 2 of all 40 characters to be displayed on the screen.

After eight rows of dots have been displayed (corresponding to row select inputs of 0–7), a complete line of 40 characters will have been displayed. To display the next line of 40 characters, the same process is repeated using 40 new bytes in the TV RAM. These 40 bytes will contain the ASCII codes for the 40 characters to be displayed on line 2 of the TV screen.

To display 24 lines of 40 characters each will require $24 \times 40 = 960$ bytes of memory in which to store the ASCII codes of the characters to be displayed. Changing a character on the screen is then simply a matter of changing the ASCII code that is stored in a particular memory location in the TV RAM. The next time this ASCII code is cycled to the character generator (within 1/60 second), this new character will be displayed on the screen. To use the TV RAM we must therefore know what memory locations are associated with each printing location on the screen.

The Apple II TV RAM

You saw in Chapter 3 (see Figure 3.7) that the TV RAM for storing text on the Apple II screen begins at memory location $400. You might expect that the memory addresses would increase across each row, and from row to row down the screen. This turns out not to be the case. It is true that the memory addresses increase in order across any 40-character line. However, the addresses do not continue in order from the end of one line to the beginning of the next.

The addresses corresponding to the first printing position in each of the 24 text lines are shown in Figure 9.3. The addresses in row 0 go from $400 to $427. However, the beginning of row 1 starts with the address $480 and not $428. In fact, address $428 corresponds to the beginning of row 8. Row 8 ends at the address $44F. The next memory location, $450,

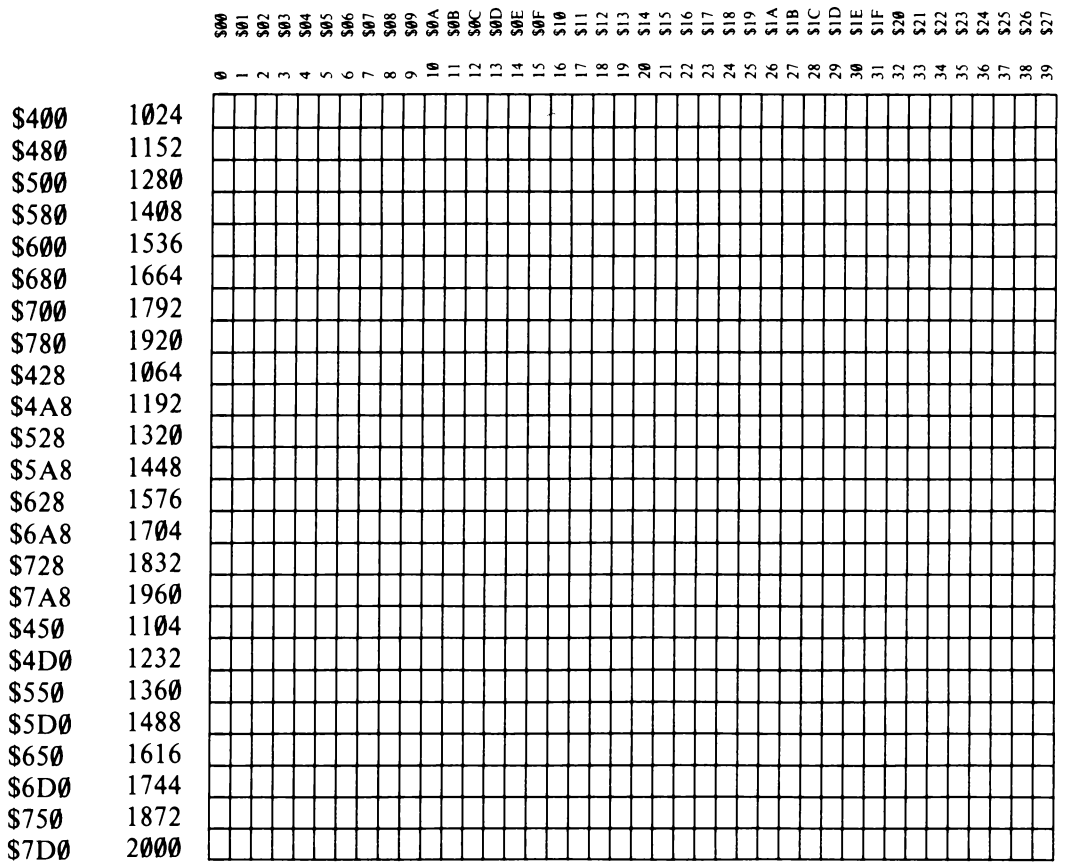


FIGURE 9.3. Memory map for TV RAM associated with the text screen.

is the starting address of row 16 ($10). This row ends with memory location $477. The next memory location corresponding to a screen printing location is $480, at the beginning of row 1. The 8 bytes between $478 and $47F do not correspond to any character positions on the screen. As we will see in Chapter 13, these 8 bytes can be used by one of the I/O slots in the Apple II.

Given a row number between 0 and 23 ($00–$17), how can you find the TV RAM address of the leftmost character on that line. These addresses are given in Figure 9.3. The binary representations of these addresses together with the binary representations of the decimal row numbers 0–23 are shown in Figure 9.4. Notice that the bit patterns in the columns labeled A, B, C, D, and E are identical for a given column label. For example, all columns labeled A contain 16 0s followed by 8 1s.

Figure 9.4 shows that given a row number with the binary bit pattern 000ABCDE, the 16-bit TV RAM address of the first print position on the line is given by 0000 01CD EABA B000. The Apple II has a built-in routine called BASCLC at location $FBC1 that will compute this starting

Screen Line No.		*Address of 1st Character on Line*	
	ABCDE		CDEABAB
0	00000000	400	0000010000000000
1	00000001	480	0000010010000000
2	00000010	500	0000010100000000
3	00000011	580	0000010110000000
4	00000100	600	0000011000000000
5	00000101	680	0000011010000000
6	00000110	700	0000011100000000
7	00000111	780	0000011110000000
8	00001000	428	0000010000101000
9	00001001	4A8	0000010010101000
10	00001010	528	0000010100101000
11	00001011	5A8	0000010110101000
12	00001100	628	0000011000101000
13	00001101	6A8	0000011010101000
14	00001110	728	0000011100101000
15	00001111	7A8	0000011110101000
16	00010000	450	0000010001010000
17	00010001	4D0	0000010011010000
18	00010010	550	0000010101010000
19	00010011	5D0	0000010111010000
20	00010100	650	0000011001010000
21	00010101	6D0	0000011011010000
22	00010110	750	0000011101010000
23	00010111	7D0	0000011111010000

FIGURE 9.4. Relationship between screen row number and starting TV RAM address.

TV RAM address for any row number stored in accumulator A. The resulting address will be stored in the two locations $28 and $29, called BASL and BASH. This routine is given in Figure 9.5; you can see how it works by comparing each step with the desired result given in Figure 9.4.

```
BASCLC   PHA              ABCDE      C
         LSR              ABCD       E
         AND   #$03        CD
         ORA   #$04       1CD
         STA   BASH
         PLA              ABCDE
         AND   #$18        AB000
         BCC   BSCLC2
         ADC   #$7F       E00AB000 ◄
BSCLC2   STA   BASL       E00AB000
         ASL
         ASL              0AB00000
         ORA   BASL       EABAB000
         STA   BASL
         RTS
```

FIGURE 9.5. Subroutine BASCLC ($FBC1) used to calculate starting TV RAM address BASL, BASH for row number A.

PRINTING A CHARACTER ON THE SCREEN

The Apple II uses locations $24 and $25 to store the horizontal and vertical cursor positions CH and CV, as shown in Figure 9.6. To calculate the address of the leftmost column in row CV, execute

```
A5 25     LDA CV
20 C1 FB  JSR BASCLC
```

The address will be in BASL and BASH at $28 and $29.

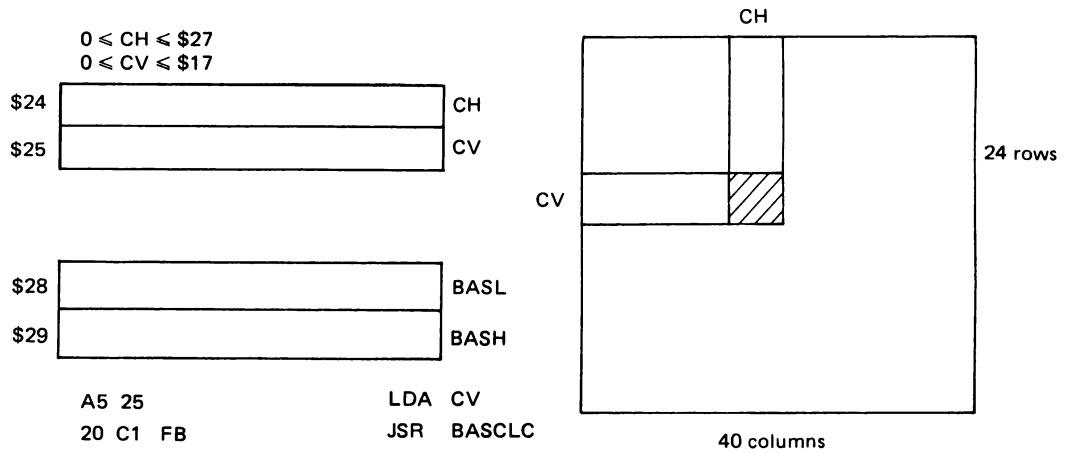


FIGURE 9.6. Apple II screen display.

To store a character at (CH,CV), calculate BASL and BASH as shown, put the ASCII code for the character to be displayed in accumulator A, and execute

```
A4 24   LDY CH
91 28   STA (BASL),Y
```

Note the use of the postindexed, indirect addressing mode, which adds the value in Y (CH) to the address stored in BASL,BASH.

As an example, the program shown in Figure 9.7 will display the letter H in column 5 of row 22 ($15). This is the row just above the command line on the screen. Type in this program and execute it starting at location $300. You will not be able to single step through this program and observe any intermediate results because the TUTOR monitor is constantly changing the values in BASL and BASH as it displays new values at different locations on the screen.

```
0300  A9 15      LDA #$15        ;Row $15
0302  20 C1 FB   JSR BASCLC      ;Calc BASL,BASH
0305  A9 C8      LDA #$C8        ;H
0307  A0 05      LDY #$05        ;Column 5
0309  91 28      STA (BASL),Y    ;Print character
030B  00         BRK
```

FIGURE 9.7. Program to print an H in column 5 of row 22.

PRINTING A MESSAGE

Suppose that you want to print a message, consisting of a string of characters, at some arbitrary location on the screen. The first thing to do is to store the message as a sequence of ASCII codes in consecutive memory locations. The message can be printed on the screen by moving this block of ASCII codes into the proper area of the TV RAM.

The ASCII codes used by the Apple II for screen display are given in Figure 9.8. Note that the normal (white on black) letter A has the code $C1 and the inverse video (black on white) letter A has the code $01. All of the characters with normal codes between $C0 and $DF have their inverse codes between $00 and $1F. Similarly, all of the characters with normal codes between $A0 and $BF have their inverse codes between $20

	Inverse				Flashing				Normal (Control)						(Lowercase)	
Decimal	0	16	32	48	64	80	96	112	128	144	160	176	192	208	224	240
Hex	$00	$10	$20	$30	$40	$50	$60	$70	$80	$90	$A0	$B0	$C0	$D0	$E0	$F0
0 $0	@	P		0	@	P		0	@	P		0	@	P		0
1 $1	A	Q	!	1	A	Q	!	1	A	Q	!	1	A	Q	!	1
2 $2	B	R	"	2	B	R	"	2	B	R	"	2	B	R	"	2
3 $3	C	S	#	3	C	S	#	3	C	S	#	3	C	S	#	3
4 $4	D	T	$	4	D	T	$	4	D	T	$	4	D	T	$	4
5 $5	E	U	%	5	E	U	%	5	E	U	%	5	E	U	%	5
6 $6	F	V	&	6	F	V	&	6	F	V	&	6	F	V	&	6
7 $7	G	W	'	7	G	W	'	7	G	W	'	7	G	W	'	7
8 $8	H	X	(	8	H	X	(	8	H	X	(	8	H	X	(	8
9 $9	I	Y	)	9	I	Y	)	9	I	Y	)	9	I	Y	)	9
10 $A	J	Z	*	:	J	Z	*	:	J	Z	*	:	J	Z	*	:
11 $B	K	[	+	;	K	[	+	;	K	[	+	;	K	[	+	;
12 $C	L	\	,	<	L	\	,	<	L	\	,	<	L	\	,	<
13 $D	M	]	-	=	M	]	-	=	M	]	-	=	M	]	-	=
14 $E	N	^	.	>	N	^	.	>	N	^	.	>	N	^	.	>
15 $F	O	_	/	?	O	_	/	?	O	_	/	?	O	_	/	?

FIGURE 9.8. Apple II ASCII codes used for screen display.

and $3F. Note that if any normal code between $A0 and $DF is ANDed with $3F, the corresponding inverse code is obtained. Therefore, messages will always be stored using their normal codes. If an inverse video message is desired, each ASCII code will be ANDed with $3F before being stored in the TV RAM.

The subroutine shown in Figure 9.9 will write a message on the screen starting at location (CH,CV). The number of bytes in the message must be stored in LENM ($FD) and the starting address of the message must be stored in MAL ($FE) and MAH ($FF). The memory location INVFLG ($32) will contain $FF for normal printing and $3F for inverse video. This subroutine is used by the TUTOR monitor and is stored at location $806E.

You can list this subroutine on the screen. First go to location $806E by typing >806E and then type /L. The command line will display

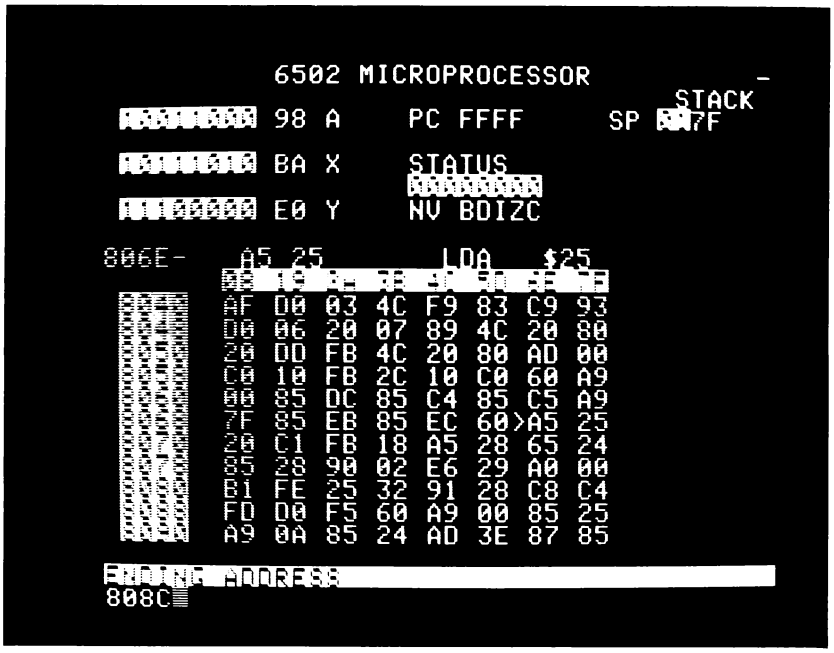
LIST: P B S

If you press P (for page), a page of disassembled code will be listed on the screen. You should be able to recognize the subroutine in Figure 9.9, starting at location $806E. Press the forward key → . Note that a new page of disassembled code is displayed. You can continue to press the → key to list a new page. Press the RETURN key to go back to the TUTOR monitor.

```
MESS    LDA   CV
        JSR   BASCLC
        CLC
        LDA   BASL
        ADC   CH
        STA   BASL
        BCC   MSS1
        INC   BASH
MSS1    LDY   $00
MSS2    LDA   (MAL),Y
        AND   INVFLG
        STA   (BASL),Y
        INY
        CPY   LENM
        BNE   MSS2
        RTS
```

FIGURE 9.9. Subroutine MESS will write a message of length LENM stored at MAL, MAH, starting at location CH, CV on the screen.

If you want to list only the MESS subroutine on the screen, go to location $806E by typing >806E and then press /LB. You will be asked for an ending address, as shown in Figure 9.10a. Type in 808C. This is the first address following the end of the subroutine. When you press RETURN, only the disassembled subroutine will be displayed on the screen, as shown in Figure 9.10b. Press RETURN to return to the TUTOR monitor.



(a)

(b)

FIGURE 9.10. /LB will disassemble a block of memory.

If you press /LS, you can scroll through a program by disassembling one line at a time. When you type /LS, the op-code at the position cursor location is disassembled and displayed on the top of the screen. Pressing the → key will cause the next instruction to be disassembled. You can continue pressing the → key to disassemble more instructions. Press the RETURN key to return to the TUTOR monitor.

As an example of using the MESS subroutine, the program given in Figure 9.11 will print the message THE END near the center of the line above the command line. The length of the message is 7. Store this value in location $300. Then store the ASCII codes for the message in locations $301–$307 by pressing /MA and typing THE END. The starting address of the message ($301) is stored in locations $308 and $309.

```
0300    07              L1      DFD     07
0301    (enter THE END
        using /MA)      M1      DFD     "THE END"
0308    01  03          MV1     EQD     M1

0310    A9  15                  LDA     #$15
0312    85  25                  STA     CV
0314    A9  10                  LDA     #$10
0316    85  24                  STA     CH
0318    AD  00  03              LDA     L1
031B    85  FD                  STA     LENM
031D    AD  08  03              LDA     MV1
0320    85  FE                  STA     MAL
0322    AD  09  03              LDA     MV1 + 01
0325    85  FF                  STA     MAH
0327    20  6E  80              JSR     MESS
032A    00                      BRK
```

FIGURE 9.11. Program to print the message THE END.

The assembler directive DFD in Figure 9.11 is used to define hex or ASCII data. For example, the statement

```
L1  DFD 07
```

defines memory location L1 ($300) to have the value 07. The statement

```
M1  DFD "THE END"
```

will cause the ASCII codes for each letter in the words "THE END" to be stored in memory starting at memory location M1 ($301).

The assembler directive EQD in the statement

```
MV1  EQD M1
```

equates the label MV1 with the address of M1. Thus, in Figure 9.11, memory locations $308 (MV1) and $309 (MV1 + 01) will contain the address of M1 ($0301) in low-byte, high-byte format.

The program itself starts at location $310. The row number $15 is stored in CV ($25) and the column number $10 is stored in CH ($24). The length value is stored in LENM ($FD), and the starting address of the message ($0301) is stored in MAL ($FE) and MAH ($FF). The message is then displayed on the screen by jumping to the subroutine MESS at $806E. The BRK instruction following JSR MESS will stop execution of the program. Type in this program and execute it, starting at location $0310. The result should be as shown in Figure 9.12.

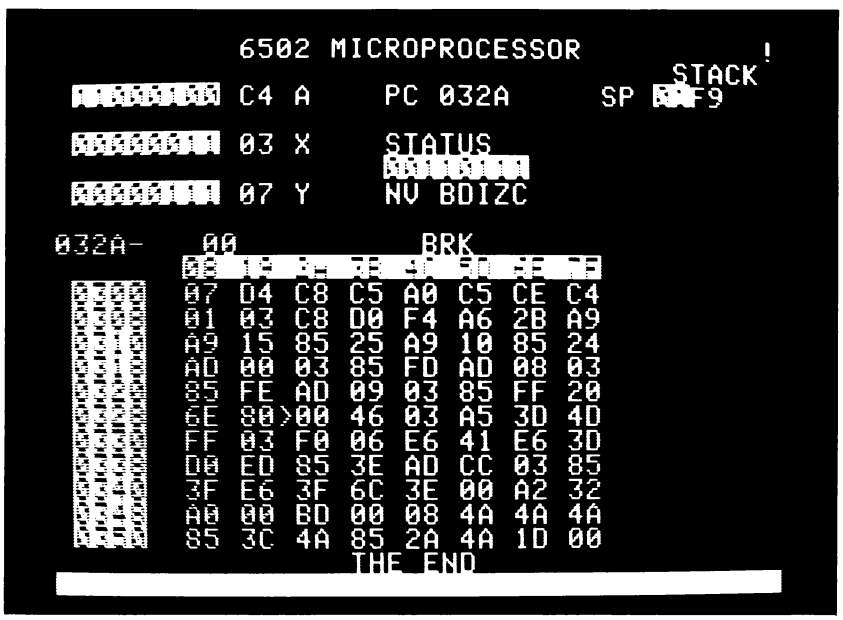


FIGURE 9.12. Result of running the program in Figure 9.11.

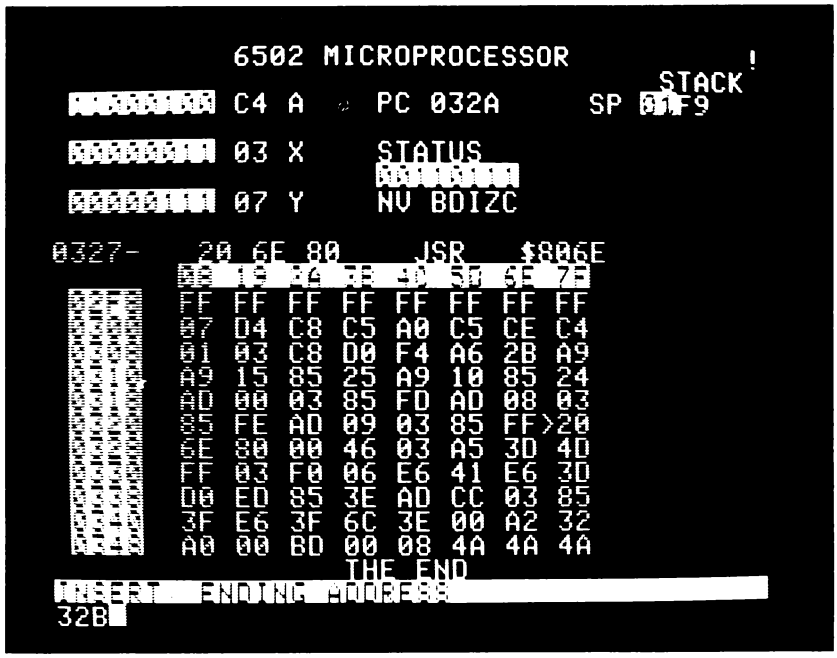
To print the message THE END in inverse video you will need to add the statements

```
A9 3F LDA  #$3F
85 32 STA  INVFLG
```

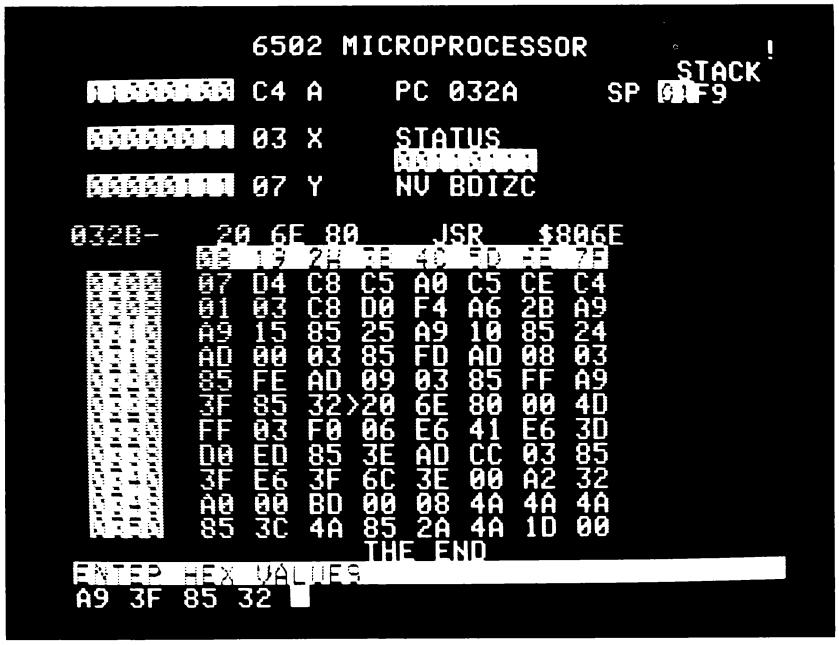
just *before* the statement JSR MESS, plus the statements

```
A9 FF LDA  #$FF
85 32 STA  INVFLG
```

just *after* the statement JSR MESS.



(a)



(b)

FIGURE 9.13. /I can be used to insert bytes in a program.

This is easy to do by using the INSERT command of the TUTOR. Go to location $327, which contains the op-code for JSR MESS. Now type /I. The command line will read

INSERT: ENDING ADDRESS

as shown in Figure 9.13a. Enter the address 32B, which is the first address after the end of the program. The command line will now read

ENTER HEX VALUES

Type in the four bytes

A9 3F 85 32

followed by RETURN. These bytes will be inserted just before the JSR MESS (20 6E 80) instruction as shown in Figure 9.13b.

Using the same technique, insert the 4 bytes A9 FF 85 32 just before the BRK instruction 00 at the end of the program.

If you now run the program starting at $310 you should obtain the result shown in Figure 9.14.

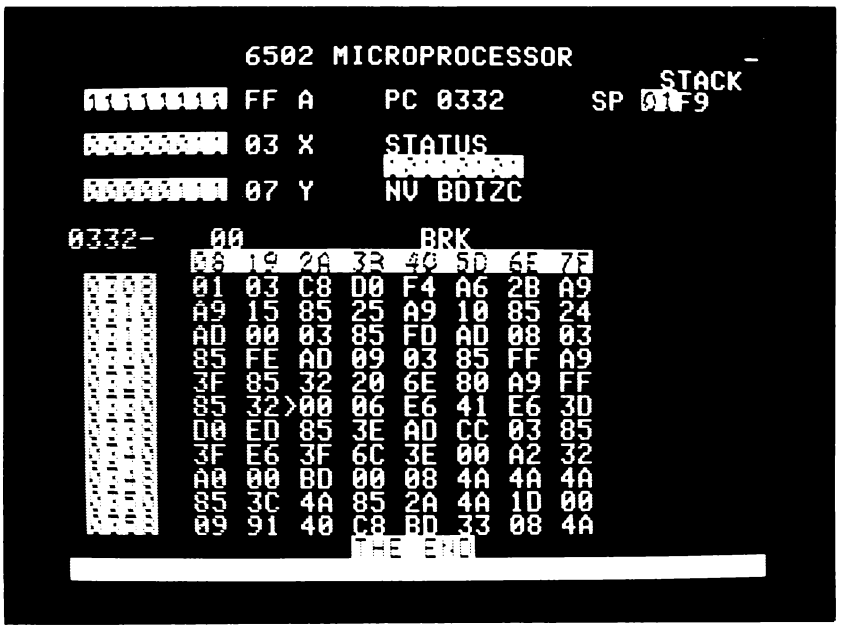


FIGURE 9.14. Printing a message using inverse video.

To return to normal video you can change the $3F in location $328 to $FF. Alternatively, you can delete the 4 bytes A9 3F 85 32, starting at location $327. You can do this using the TUTOR command DELETE. Go to location $327 and type /D. The command line will read

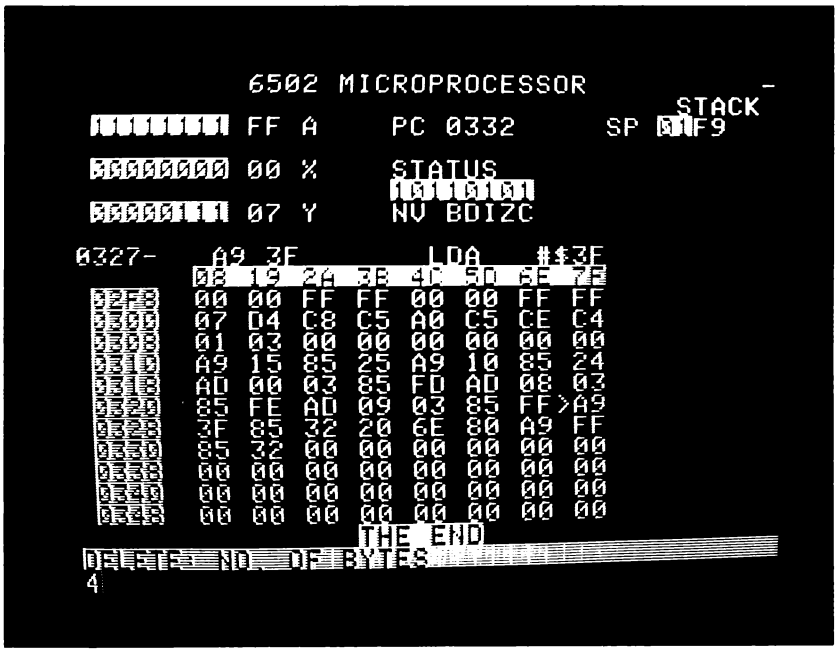
DELETE: NO. OF BYTES

as shown in Figure 9.15a. Type 4 followed by RETURN. The command line will read

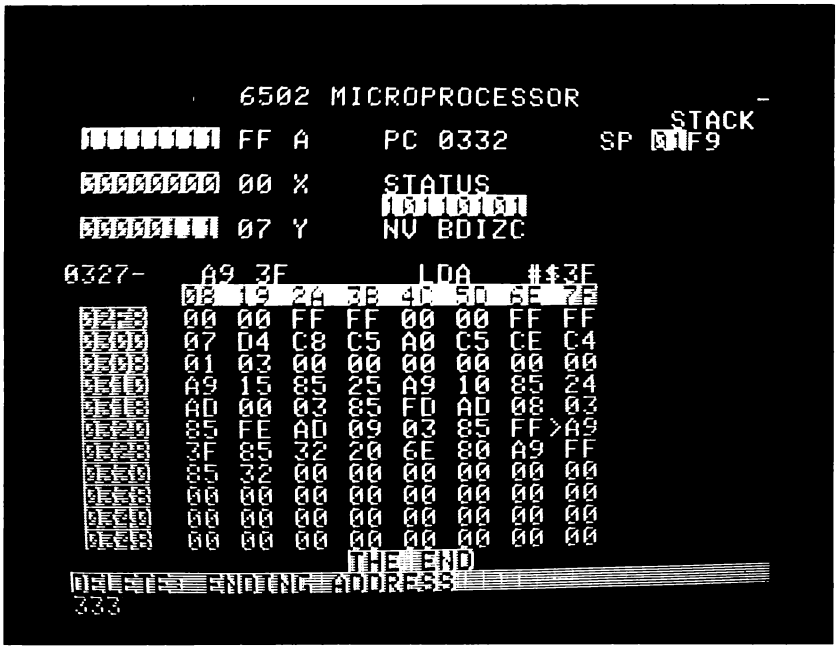
DELETE: ENDING ADDRESS

as shown in Figure 9.15b. Type 333, which is the first address after the end of the program. This will determine the last byte that is "moved up" when the 4 bytes are deleted. When you press RETURN, the 4 bytes will be deleted and all bytes up to location $333 will be moved up 4 bytes (see Figure 9.15c.). Try executing the program again starting at location $310.

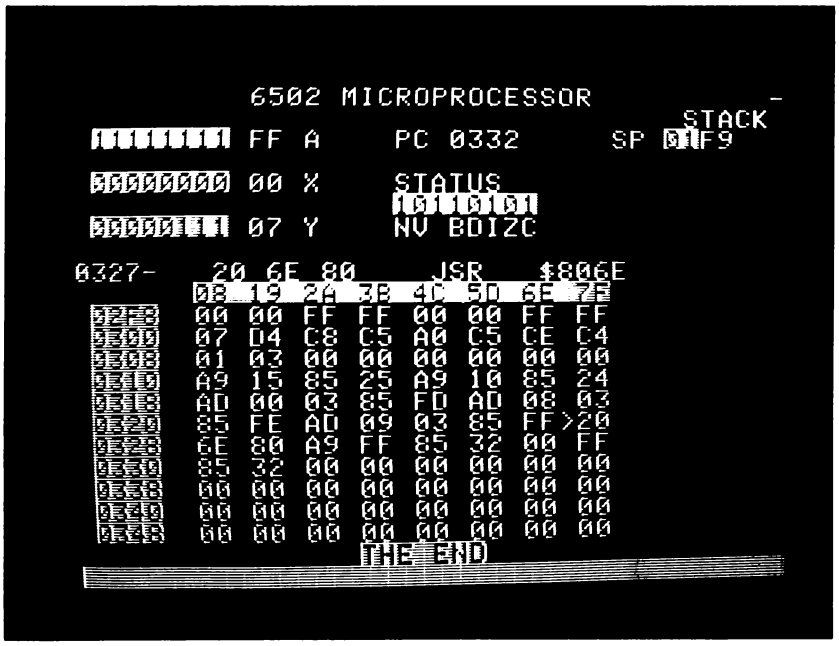
FIGURE 9.15 (below and opposite). /D can be used to delete bytes.



(a)


6502 MICROPROCESSOR
STACK
FF A
PC 0332
SP 01F9
00 X
STATUS
07 Y
NV BDIZC
0327- A9 3F LDA #$3F
08 19 2A 3B 4D 50 6E 7E
0328 00 00 FF FF 00 00 FF FF
0300 07 D4 C8 C5 A0 C5 CE C4
0308 01 03 00 00 00 00 00 00
0310 A9 15 85 25 A9 10 85 24
0318 AD 00 03 85 FD AD 08 03
0320 85 FE AD 09 03 85 FF>A9
0328 3F 85 32 20 6E 80 A9 FF
0330 85 32 00 00 00 00 00 00
0338 00 00 00 00 00 00 00 00
0340 00 00 00 00 00 00 00 00
0348 00 00 00 00 00 00 00 00
THE END
DELETE: ENDING ADDRESS
333

(b)


6502 MICROPROCESSOR
STACK
FF A
PC 0332
SP 01F9
00 X
STATUS
07 Y
NV BDIZC
0327- 20 6E 80 JSR $806E
08 19 2A 3B 4D 50 6E 7E
0328 00 00 FF FF 00 00 FF FF
0300 07 D4 C8 C5 A0 C5 CE C4
0308 01 03 00 00 00 00 00 00
0310 A9 15 85 25 A9 10 85 24
0318 AD 00 03 85 FD AD 08 03
0320 85 FE AD 09 03 85 FF>20
0328 6E 80 A9 FF 85 32 00 FF
0330 85 32 00 00 00 00 00 00
0338 00 00 00 00 00 00 00 00
0340 00 00 00 00 00 00 00 00
0348 00 00 00 00 00 00 00 00
THE END

(c)

The Apple II has a number of built-in subroutines related to the text screen display. We will look at how to use routines to clear the screen, print a character on the screen, and print hex values.

Clearing the Screen

The subroutine HOME at $FC58 will clear the screen inside the window shown in Figure 9.16. The 4 bytes $20–$23 define the size of the window as given in Figure 9.16. The default values are the ones shown, corresponding to the full screen.

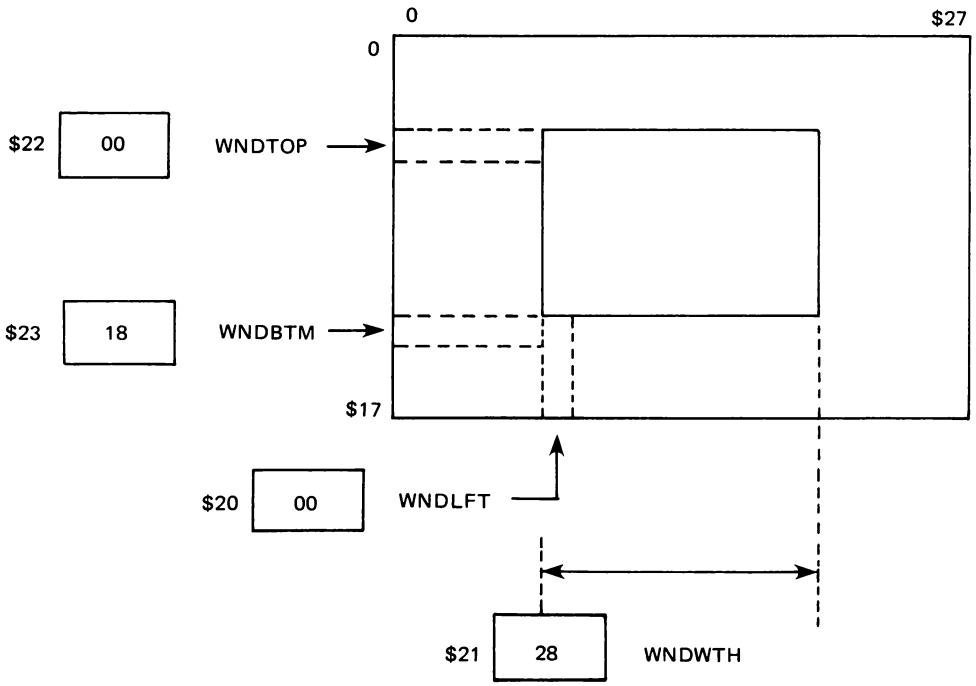


FIGURE 9.16. JSR HOME ($FC58) will clear the screen inside window.

You can test this subroutine by typing in and executing the statements shown in Figure 9.17.

```
0300 20 58 FC JSR HOME
0303 20 56 80 JSR KEYIN
0306 4C 00 80 JMP TUTOR
```

FIGURE 9.17. This program will clear the screen until any key is pressed.

The subroutine KEYIN is the TUTOR routine shown in Figure 9.18 that waits for a key to be pressed and then returns with the ASCII code for the key in accumulator A. When you run the program in Figure 9.17 the screen will clear until you press any key. Try it. When you press a key the statement JMP TUTOR will jump to the start of the TUTOR monitor at location $8000.

```
8056    AD 00 C0    KEYIN   LDA   KEY
8059    10 FB               BPL   KEYIN
805B    2C 10 C0            BIT   KEYSTB
805E    60                  RTS
```

FIGURE 9.18. This TUTOR routine KEYIN ($8056) will wait for a key to be pressed and return with the ASCII code for the key in A.

Printing a Character

The subroutine COUT1 at location $FDF0 prints the character whose ASCII code is in accumulator A at the horizontal position CH, using the current line address that is in BASL,BASH. If the value of the vertical cursor position CV changes, you must recalculate BASL,BASH using

```
LDA CV
JSR BASCLC
```

The subroutine COUT1 advances the cursor and handles RETURN ($8D), linefeed ($8A), bell ($87), and backspace ($88).

As an example of using COUT1, type in the program shown in Figure 9.19 and execute it. This program will clear the screen and then wait for you to press a key using the TUTOR subroutine KEYIN (see Figure 9.18). If you press any key other than the exclamation point it will be printed on the screen using COUT1. Try typing lots of keys. Note that the RETURN, linefeed, and backspace keys work. Press CTRL G to ring the bell. Press the ! key to return to the TUTOR monitor.

```
0300    20 58 FC    TYPE    JSR HOME      ;clear screen
0303    20 56 80    T1      JSR KEYIN     ;wait for key input
0306    C9 A1               CMP $A1       ;if key is !
0308    D0 03               BNE T2        ;then
030A    4C 00 80            JMP TUTOR     ;return to TUTOR
030D    20 F0 FD    T2      JSR COUT1     ;else print character
0310    4C 03 03            JMP T1        ;go to KEYIN
```

FIGURE 9.19. Program to type any characters on the screen.

No blinking cursor is displayed in the program shown in Figure 9.19. To display a blinking cursor you must print a flashing space (ASCII code $60) on the screen. The built-in Apple II routine RDKEY at location $FD0C

does this and then behaves similarly to the KEYIN routine in Figure 9.18. Change the statement at address $303 in Figure 9.19 to

```
20 0C FD    T1 JSR   RDKEY
```

and rerun the program. It should behave the same as it did before except for the inclusion of a blinking cursor.

Printing Hex Values

The subroutine PRHEX at $FDE3 will print the lower nibble of accumulator A as a hex digit. For example, if accumulator A contains the value $3B, the statement JSR PRHEX will print a B on the screen at the current print position. This routine is shown in Figure 9.20. After masking the upper nibble of A by ANDing A with $0F, the ASCII code of the remaining hex digit must be found. This can be done with the following algorithm (see Figure 9.8):

```
ORA #$B0
if A >= $BA
then add $07
```

Note that in Figure 9.20 the statement ADC #$06 will really add #$07 to A because the carry will always be set.

```
PRHEX   AND #$0F      ;mask upper nibble
        ORA #$B0      ;Hex-ASCII
        CMP #$BA      ;conversion
        BCC COUT
        ADC #$06
COUT    JMP (CSWL)              CSWL  F0 $36
                                      FD $37
```

FIGURE 9.20. PRHEX at $FDE3 will print the lower nibble of A as a hex digit.

The last statement in Figure 9.20 is an indirect JUMP to the address stored in locations $36 and $37. The address stored here is normally $FDF0, the address of COUT1. We will discuss the use of locations $36 and $37 in Chapter 13.

As an example of using PRHEX, type in the program shown in Figure 9.21. Single step through this program. Note that single stepping a JSR to any built-in routine will cause the entire routine to be executed; the program will stop at the instruction following JSR. When you execute JSR PRHEX in Figure 9.21, the B in accumulator A should be printed on line $15 (the line above the command line).

```
0300    A9 15       LDA  #$15
0302    20 C1 FB    JSR  BASCLC
0305    A9 03       LDA  #$03
0307    85 24       STA  CH
0309    A9 3B       LDA  #$3B
030B    20 E3 FD    JSR  PRHEX
030E    00          BRK
```

FIGURE 9.21. Program to print the B from $3B on line $15.

The built-in subroutine PRBYTE at $FDDA will print the byte in A as two hex digits. For example, change the instruction at address $30B in Figure 9.21 to

```
030B  20 DA FD        JSR PRBYTE
```

and single step through the program again. Note that the entire byte $3B is printed this time.

The built-in subroutine PRNTAX at $F941 prints the contents of A and X as a four-digit hex number. For example, if A contains $3A and X contains $C7, then JSR PRNTAX will print the characters 3AC7 at the current printing location on the screen.

EXERCISE 9.1

Write a program that will display on line $15 the address stored in locations $FE and $FF, followed by the contents of that address.

EXERCISE 9.2

Write a program that will print your name on line $15 of the screen.

EXERCISE 9.3

Write a program that will display on line $15 the word ONE, TWO, or THREE depending upon whether you press key 1, 2, or 3, respectively.

10

Low-Resolution Graphics

You learned how to display text on the screen in Chapter 9. In addition to displaying text by storing ASCII codes in the TV RAM, it is also possible to display graphic figures. Low-resolution graphics will be described in this chapter, and high-resolution graphics will be discussed in Chapter 11. In this chapter you will learn

1. to switch to the low-resolution graphics mode
2. to plot spots in various colors
3. to plot horizontal and vertical lines
4. to fill the screen with various colors

SCREEN SOFT SWITCHES

The 6502 microprocessor used in the Apple II computer can address a maximum of 64K (65,536) bytes of memory. This memory space is divided among read–write, or random-access memory (RAM), input/output (I/O) memory, and read only memory (ROM). Figure 10.1 shows how this memory is allocated for an Apple II Plus computer with 48K bytes of RAM. Notice from the left-hand and right-hand sides of the figure that locations $0000–$BFFF are the 48K bytes of RAM, locations $C000–$CFFF are 4K bytes associated with I/O, and locations $D000–$FFFF are 12K bytes of ROM.

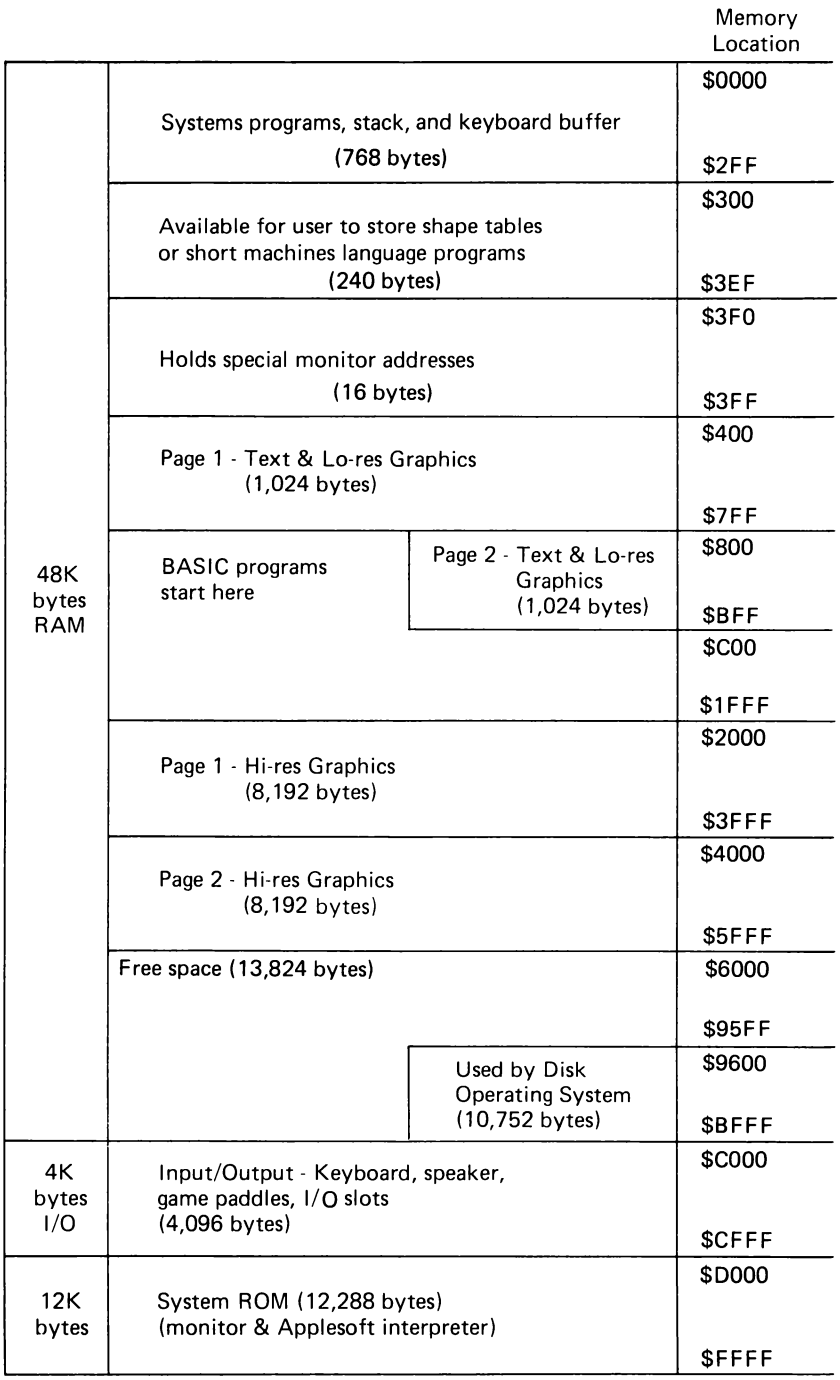


FIGURE 10.1. Memory map of an Apple II Plus with 48K of RAM.

As described in Chapter 9, memory locations $400–$7FF are used to store the ASCII codes that are being displayed on the TV screen. These same memory locations are used to store the color codes used to plot various colored spots in the low-resolution graphics mode. These memory locations ($400–$7FF) are referred to as page 1 (or the primary page) of text and low-resolution (lo-res) graphics. It turns out that there is a page 2 (or secondary page) of text and low-resolution graphics at memory locations $800–$BFF.

When you use high-resolution graphics (to be described in Chapter 11), the information displayed on the screen is stored in memory at locations $2000–$3FFF. This is called page 1 (or the primary page) of high-resolution (hi-res) graphics. There is also a page 2 (or secondary page) of high-resolution graphics at memory locations $4000–$5FFF.

You can switch between these various pages by using the "soft switches" shown in Figure 10.2. There are four switches (shown on the left side of the figure); the position of each switch is associated with a particular memory address. Note that the addresses of these soft switches are in the I/O area of memory. To turn a switch to a particular position you simply have to refer to the address, using any statement such as LDA *addr* or BIT *addr*.

There are 10 different modes that you can enable by setting the four soft switches. These ten modes are illustrated by the tree graph shown in Figure 10.2. The top of the tree is switch 1 (addresses $C051 and $C050), which selects either *text* or *graphics*. Row 2 of the tree uses switch 2 (address $C054 and $C055) to select either page 1 or page 2. Row 3 of the tree uses switch 3 (addresses $C053 and $C052) to select between mixed text and graphics (that is, four lines of text at the bottom of the screen) and all graphics. The bottom row of the tree uses switch 4 (addresses $C056 and $C057) to select between low-resolution and high-resolution graphics.

To illustrate the use of the soft switches, type in the program shown in Figure 10.3. Execute the program starting at location $310. The screen should switch to the low-resolution graphics mode. Pressing key B will switch back and forth between the text mode (switch $C051) and the graphics mode (switch $C050). Press any other key to return to the TUTOR monitor. Executing the TUTOR monitor program MONIT at location $8000 will switch the soft switches to the text mode. The statement CMP #"B in Figure 10.3 means compare the contents of accumulator A with the ASCII code for the letter B.

You should notice some correspondence between the text displayed in the text mode and the graphics displayed in the graphics mode. This is because the same RAM contents are being displayed in each case, but in a different format. We will see what this correspondence is in the next section.

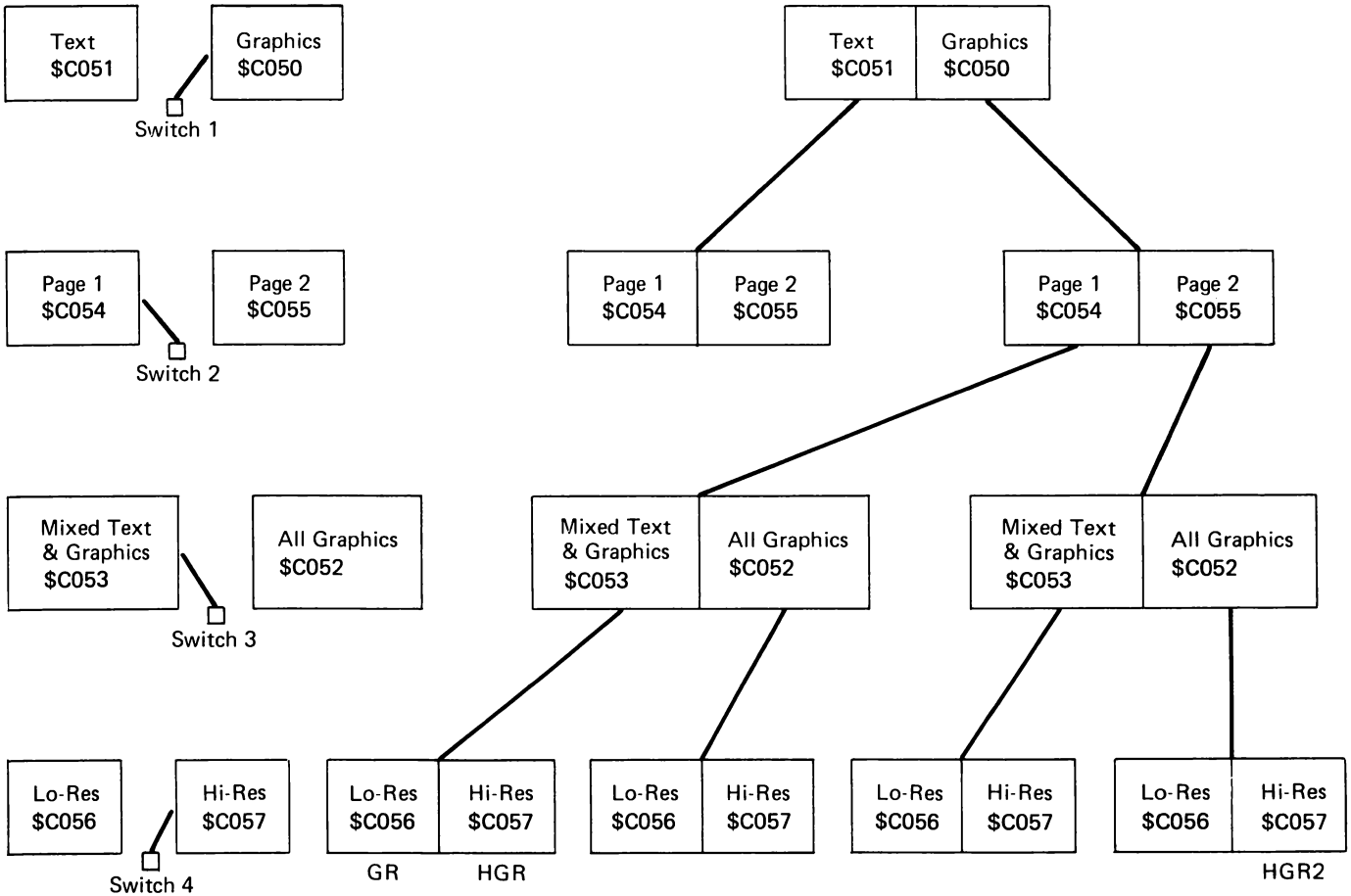


FIGURE 10.2. Soft switches used to select text and graphics pages.

```
0310  2C 50 C0    AGAIN   BIT   $C050    ;switch to graphics
0313  20 56 80            JSR   KEYIN    ;wait for key
0316  C9 C2               CMP   #"B      ;if B
0318  D0 0A               BNE   DONE     ;then
031A  2C 51 C0           BIT   $C051    ;switch to text
031D  20 56 80           JSR   KEYIN    ;wait for key
0320  C9 C2              CMP   #"B      ;if B
0322  F0 EC              BEQ   AGAIN    ;then switch again
0324  4C 00 80    DONE    JMP   MONIT    ;go to TUTOR
```

FIGURE 10.3. Program to switch between text and graphics modes.

To see the effect of soft switch number 3, add the statement

```
030D  2C 50 C0   BIT $C050   ;switch to graphics
```

to the beginning of the program in Figure 10.3. Then change the two statements at $310 and $31A to

```
0310  2C 52 C0   BIT $C052   ;switch to all graphics
031A  2C 53 C0   BIT $C053   ;switch to mixed text and graphics
```

and execute the new program starting at $30D. Pressing key B should switch between all graphics and mixed text and graphics. Press any other key to return to the TUTOR monitor.

To see the effect of soft switch number 2, change the two statements at $310 and $31A to

```
0310 2C 55 C0 BIT $C055 ;switch to page 2
031A 2C 54 C0 BIT $C054 ;switch to page 1
```

and execute the new program starting at $30D. Pressing key B should switch between page 1 ($400–$7FF) and page 2 ($800–$BFF). Press any other key to return to the TUTOR monitor.

Clearing the Screen

After switching to the low-resolution graphics mode you can clear the screen by calling a built-in subroutine. The routine CLRSCR at location $F832 will clear the entire low-resolution screen (you should be in the all-graphics mode). The routine CLRTOP at location $F836 will clear the top part of the low-resolution screen, leaving the four lines of text at the bottom.

The built-in routine SETGR at location $FB40 switches to graphics ($C050), switches to mixed text and graphics ($C053), clears the top of the low-resolution screen, sets the text window to the bottom four lines of the screen, and moves the cursor to the bottom line of the screen. The

Apple II sets the LO-RES switch ($C056) and the page 1 switch ($C054) when the INIT routine at location $FB2F is called during the RESET operation. Therefore, if you call SETGR from the text mode without having changed the low-res/hi-res soft switch 4, you will switch to the "normal" low-resolution graphics mode. However, if you have been in high-resolution graphics or on page 2, you may not get what you expect by calling SETGR. It may then be better to set the four soft switches yourself. The routine SETTXT at location $FB39 will switch to the text mode ($C051) and change the text window to full screen. This routine is also called by INIT at location $FB2F.

LOW-RESOLUTION COLORS

Sixteen colors are available in low-resolution graphics. They are assigned numbers between $00 and $0F as shown in Table 10.1. Before a spot can be plotted on the screen, a color number must be loaded in accumulator A and the set color subroutine SETCOL at $F864 must be called. All subsequent plot routines will plot spots of this color until SETCOL is called again with a different color number in accumulator A.

Table 10.1 Low-Resolution Colors

$00 Black	$08 Brown
$01 Magenta	$09 Orange
$02 Dark blue	$0A Gray 2
$03 Purple	$0B Pink
$04 Dark green	$0C Light green
$05 Gray 1	$0D Yellow
$06 Medium blue	$0E Aqua
$07 Light blue	$0F White

Recall from Chapter 9 that each character displayed on the screen has an ASCII code that is stored in a certain memory location corresponding to a particular screen location. For example, the letter S will have the ASCII code D3 stored in the TV RAM.

When the screen is switched to the low-resolution graphics mode, the bytes stored in the TV RAM are interpreted in a different way. The screen area occupied by the ASCII character is divided into two low-resolution spots, one on top of the other. The low-order nibble (4 bits) of the memory byte is the color number of the top spot and the high-order nibble is the color number of the bottom spot, as shown in Figure 10.4. Note that the ASCII code D3 is reinterpreted as the two colors purple (3) and yellow (D), using the numbers in Table 10.1.

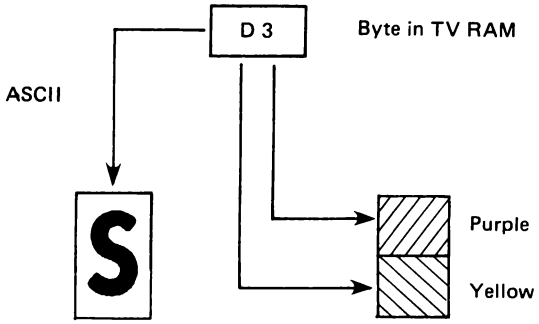


FIGURE 10.4. Relationship between ASCII characters and low-resolution graphics spots.

You should now be able to explain the colors that were displayed on the screen when you switched between the text and graphics modes using the program in Figure 10.3. For example, the command line in the TUTOR is made from a row of inverse spaces (ASCII code $20). This becomes translated (in the full-graphics mode) into a dark blue (2) and black (0) line. Blank spaces in the TUTOR with the ASCII code $A0 become translated into gray and black spots.

In the text mode the screen contains 24 rows of 40 characters each. The low-resolution graphics mode will therefore contain 48 rows of 40 spots each. In the mixed text and graphics mode, four lines of text are left at the bottom of the screen. The remaining 20 lines of text are transformed into 40 rows of low-resolution spots, resulting in a 40×40 grid for the graphics figures.

PLOTTING A SPOT

In the mixed-text and low-resolution graphics mode the screen is divided into a 40×40 grid with four lines of text at the bottom of the screen (See Figure 10.5). The column positions of the grid are numbered $00 through $27 from left to right. This is called the *x* position or *x*-coordinate. The row positions of the grid are numbered $00 through $27 from top to bottom. This is called the *y* position or *y*-coordinate. Any one of the 1,600 (40 × 40 = 1,600) small squares or blocks on the grid can be identified by giving its *x* and *y* coordinates. For example, in Figure 10.5 the shaded block is located at the coordinates *x* = $19, *y* = $0F.

You can plot a colored spot at any of the 1,600 grid positions on the screen. These spots can be one of the 16 colors given in Table 10.1. To set a color, load the color number in accumulator A and jump to the subroutine SETCOL at $F864.

To plot a spot at location *x,y*, store the *x*-coordinate in index register Y and the *y*-coordinate in accumulator A and call the subroutine PLOT at

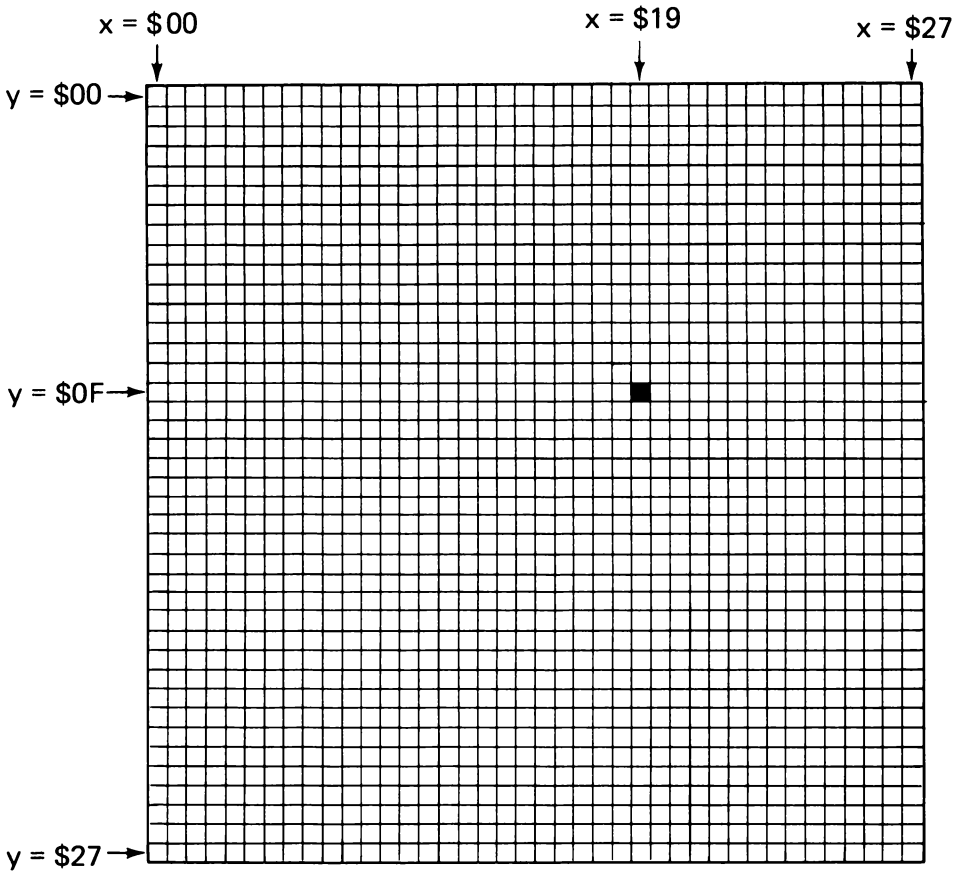


FIGURE 10.5. The low-resolution graphics mode divides the screen into a 40x40 grid.

$F800. As an example, the program shown in Figure 10.6 will plot a yellow spot at x = $19, y = $0F. Type in this program and execute it starting at $300. The 40×40 graphics screen is cleared and a single yellow spot should appear on the screen. Pressing any key will return to the text mode and the TUTOR monitor.

```
0300    20 40 FB    JSR SETGR     ;set graphics
0303    A9 0D       LDA #$0D      ;yellow
0305    20 64 F8    JSR SETCOL    ;color
0308    A9 0F       LDA #$0F      ;y-coordinate
030A    A0 19       LDY #$19      ;x-coordinate
030C    20 00 F8    JSR PLOT      ;plot spot
030F    20 56 80    JSR KEYIN     ;wait for key
0312    4C 00 80    JMP MONIT     ;return to TUTOR
```

FIGURE 10.6. Program to plot a yellow spot at x = $19, y = $0F.

EXERCISE 10.1

Write a program that will plot a diagonal blue line from the coordinate $x = \$05$, $y = \$0A$ to $x = \$0F$, $y = \$14$.

PLOTTING HORIZONTAL AND VERTICAL LINES

Horizontal and vertical lines can be plotted using the built-in routines HLIN at $F819 and VLIN at $F828. The subroutine HLIN plots a horizontal line from $x1$ to $x2$ at row y, as shown in Figure 10.7. Before calling HLIN, store the value of y in accumulator A, the value of $x1$ in index register Y, and the value of $x2$ in memory location $2C. As an example, the program in Figure 10.8 plots a red line all the way across the top of the screen. Type in this program and execute it, starting at $300. Press any key to return to the TUTOR monitor.

FIGURE 10.7.
Plotting a horizontal line.

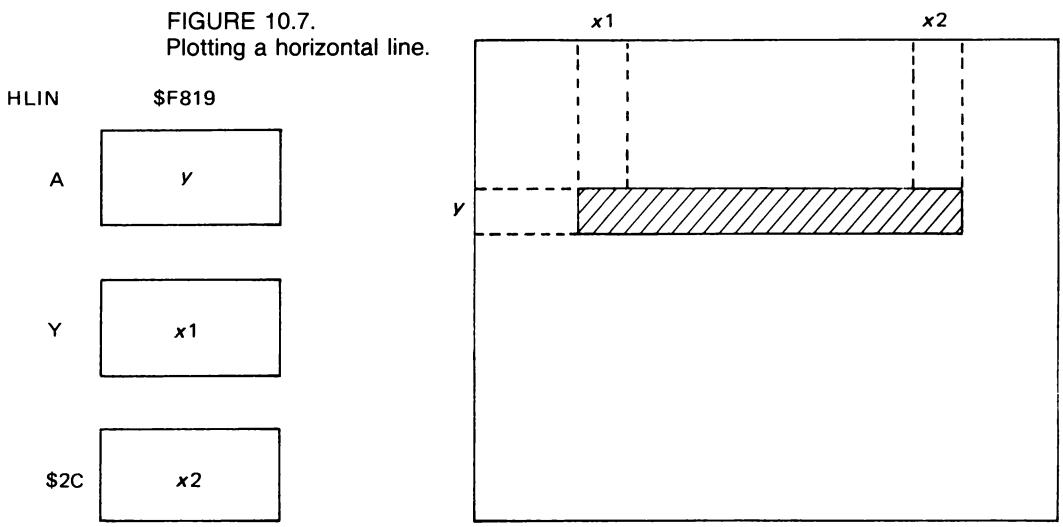
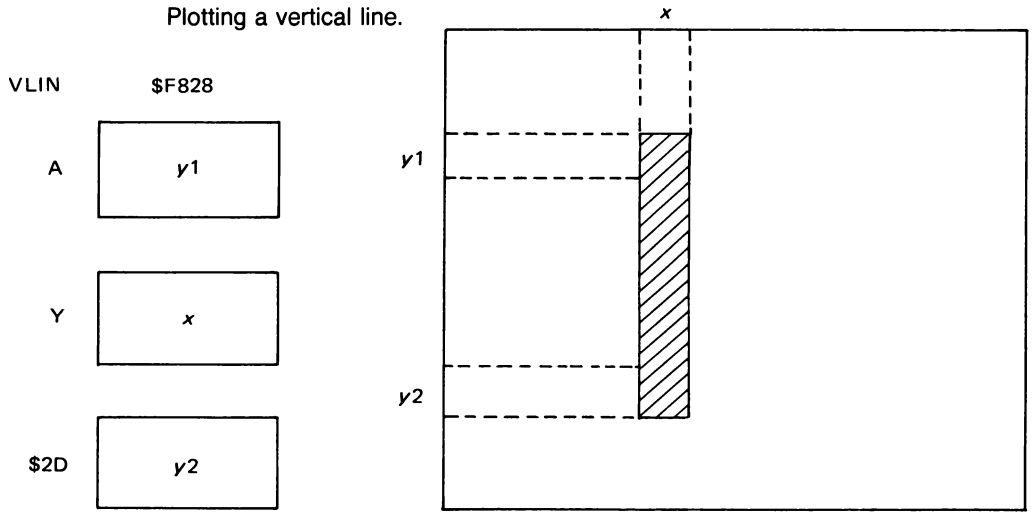


FIGURE 10.8. Program to plot a red horizontal line across the top of the screen.

```
0300    20 40 FB    JSR SETGR     ;set graphics
0303    A9 01       LDA #$01      ;red
0305    20 64 F8    JSR SETCOL    ;color
0308    A9 27       LDA #$27
030A    85 2C       STA $2C       ;x2 = $27
030C    A9 00       LDA #$00      ;y = 0
030E    A8          TAY           ;x1 = 0
030F    20 19 F8    JSR HLIN      ;plot horizontal line
0312    20 56 80    JSR KEYIN     ;wait for key
0315    4C 00 80    JMP MONIT     ;return to TUTOR
```

The subroutine VLIN plots a vertical line from $y1$ to $y2$ at column x as shown in Figure 10.9. Before calling VLIN, store the value of $y1$ in accumulator A, the value of x in index register Y, and the value of $y2$ in memory location $2D. As an example, the program in Figure 10.10 plots a green line along the right edge of the screen. Type in this program and execute it, starting at $300. Note that only a few locations are different from the program in Figure 10.8. Press any key to return to the TUTOR monitor.

FIGURE 10.9.
Plotting a vertical line.




```
0300    20 40 FB    JSR SETGR     ;set graphics
0303    A9 04       LDA #$04      ;green
0305    20 64 F8    JSR SETCOL    ;color
0308    A9 27       LDA #$27
030A    85 2D       STA #$2D      ;y2 = $27
030C    A8          TAY           ;x = $27
030D    A9 00       LDA #$00      ;y1 = 0
030F    20 28 F8    JSR VLIN      ;plot vertical line
0312    20 56 80    JSR KEYIN     ;wait for key
0315    4C 00 80    JMP MONIT     ;return to TUTOR
```

FIGURE 10.10. Program to plot a green vertical line down the right edge of the screen.

A summary of low-resolution graphics routines is given in Table 10.2. The last routine in the table can be used to find the color of any x,y location on the screen.

Table 10.2 Low-Resolution Graphics Routines

Name	*Location*	*Arguments*	*Registers Changed*	*Function*
SETGR	$FB40	None	A,Y	Set lo-res graphics mode
SETCOL	$F864	A = color no.	A	Set color number
CLRSCR	$F832	None	A,Y	Clear entire lo-res screen
CLRTOP	$F836	None	A,Y	Clear top 40 rows of low-res screen
PLOT	$F800	A = *y*-coord Y = *x*-coord	A	Plot spot at *x,y*
HLIN	$F819	A = *y*-coord Y = *x*1-coord $2C = *x*2-coord	A,Y	Plot horizontal line from *x*1 to *x*2 at row *y*
VLIN	$F828	A = *y*1-coord Y = *x*-coord $2D = *y*2-coord	A	Plot vertical line from *y*1 to *y*2 at column *x*
SCRN	$F871	A = *y*-coord Y = *x*-coord	A	Return with color of spot at *x,y* in A

SWITCHING SCREEN COLORS

The subroutine given in Figure 10.11 will fill the entire 40×40 low-resolution graphics screen with the current color by plotting 40 vertical lines starting at the right side of the screen. The program shown in Figure 10.12 will fill the screen with the color corresponding to the key pressed. Pressing the RETURN key will return to the TUTOR monitor. The subroutine ASCHEX is an ASCII-to-hex conversion routine that is in the TUTOR monitor at $83AD. It converts the ASCII value in accumulator A to its corresponding hex value. As an example, if you press key 3, accumulator A will contain the ASCII code $B3 when the program returns from KEYIN. The subroutine ASCHEX will convert this to $03. Similarly, if you press key D, then ASCHEX will convert the ASCII code $C4 to the hex value $0D. This value will be used to set the color (yellow) in SETCOL. The subroutine COLSCN given in Figure 10.11 in then called, which fills the screen with that color.

```
0300   A0 27      COLSCN   LDY #$27     ;x = $27
0302   84 2D               STY $2D      ;y2 = $27
0304   A9 00      CLS1     LDA #$00     ;y1 = 0
0306   20 28 F8            JSR VLINE    ;plot vertical line
0309   88                  DEY          ;x = x - 1
030A   10 F8               BPL CLS1     ;do entire screen
030C   60                  RTS
```

FIGURE 10.11. Subroutine to fill screen with current color.

```
0310    20 40 FB          JSR SETGR      ;set graphics
0313    20 56 80  AGAIN   JSR KEYIN      ;wait for key
0316    C9 8D             CMP #$8D       ;if RETURN
0318    F0 0C             BEQ DONE       ;then go to TUTOR
031A    20 AD 83          JSR ASCHEX     ;else convert to hex
031D    20 64 F8          JSR SETCOL     ;set color
0320    20 00 03          JSR COLSCN     ;color the screen
0323    4C 13 03          JMP AGAIN      ;do it again
0326    4C 00 80  DONE    JMP MONIT      ;go to TUTOR
```

FIGURE 10.12. Program to change screen colors.

Type in the subroutine in Figure 10.11 and the program in Figure 10.12. Execute the program starting at location $310. Pressing keys 0–F should cycle through all 16 colors. Press the RETURN key to return to the TUTOR monitor.

EXERCISE 10.2

Plot a yellow horizontal line across the bottom of the screen.

EXERCISE 10.3

Plot an orange vertical line along the left edge of the screen.

EXERCISE 10.4

Write a program to produce a red and blue vertical striped pattern that completely fills the screen.

11

High-Resolution Graphics

Chapter 10 described the use of low-resolution graphics in the Apple II. Graphic figures of much higher resolution can also be plotted using a different graphics mode. In this chapter you will learn

1. how high-resolution graphics data are stored in the Apple II memory
2. what colors can be plotted in high-resolution graphics
3. to plot a point and a line in high-resolution graphics
4. to plot graphic figures using shape tables.

HIGH-RESOLUTION MEMORY BUFFERS

There are two 8K memory buffers used to store high-resolution graphics data. The page 1 buffer starts at location $2000 and the page 2 buffer starts at location $4000, as shown in Figure 10.1 of the last chapter.

Soft switch number 4, shown in Figure 10.2, is used to switch between low-resolution and high-resolution graphics. The program shown in Figure 11.1 (which is similar to the one in Figure 10.3) will allow you to switch between low-resolution and high-resolution graphics by pressing key B. Type in this program and execute it starting at location $30D. Press any key other than B to return to the TUTOR monitor.

```
030D  2C 50 C0        BIT $C050     ;switch to graphics
0310  2C 57 C0  AGAIN BIT $C057     ;switch to hi-res
0313  20 56 80        JSR KEYIN     ;wait for key
0316  C9 C2           CMP #"B       ;if "B"
0318  D0 0A           BNE DONE      ;then
031A  2C 56 C0        BIT $C056     ;switch to lo-res
031D  20 56 80        JSR KEYIN     ;wait for key
0320  C9 C2           CMP #"B       ;if "B"
0322  F0 EC           BEQ AGAIN     ;then switch again
0324  4C 00 80  DONE  JMP MONIT     ;go to TUTOR
```

FIGURE 11.1. Program to switch between low-resolution and high-resolution graphics.

As shown in Figure 10.2, there are four different high-resolution graphics modes depending upon the settings of the page 1/page 2 switch and the mixed-text/all-graphics switch.

Each high-resolution graphics screen can plot 280 points in the horizontal direction and either 160 or 192 points in the vertical direction, depending upon the setting of the mixed-text/all-graphics switch. A point on the screen is specified by its x-coordinate (0–279) and its y-coordinate (0–191), as shown in Figure 11.2. Page 1 stores its image data in the 8K-byte memory buffer between locations $2000 and $3FFF. Page 2 of high-resolution graphics uses the memory between locations $4000 and $5FFF.

Each plot position in high resolution graphics is associated with a single bit in one of the memory buffer bytes. The seven low-order bits in each memory buffer byte contain the intensity information for seven consecutive plot positions along a row on the screen. When using a black and white monitor, a 1 in a bit position will produce a white spot on the screen, while a 0 in a bit position will produce a black spot on the screen. Bit position 0 in each byte is plotted first. For example, the value $37 stored in a byte will produce the dot pattern on the screen shown in Figure 11.3.

Only 7 bits in each byte correspond to plot positions on the screen. Forty bytes will therefore be required to plot a single row of 280 (40×7) dots. Recall that the 24 rows in the text mode plot characters that are eight dots high. We will need a separate byte for each of these eight rows of dots in order to plot 192 (24×8) rows of graphics data. Therefore, the total memory required for a high-resolution graphics display is $192 \times 40 = 7{,}680$ bytes. The memory map of page 1 of the high-resolution graphics screen is shown in Figure 11.4. Note that as in the text mode these screen addresses do not increase in order from row to row on the screen.

The built-in routine HPOSN at $F411 can be used to find the address of a given plot position. The use of this subroutine is illustrated in Figure 11.5. The sample program shown in this figure finds the address associated with the screen coordinates $x0 = 37$ ($25) and $y0 = 0$. After

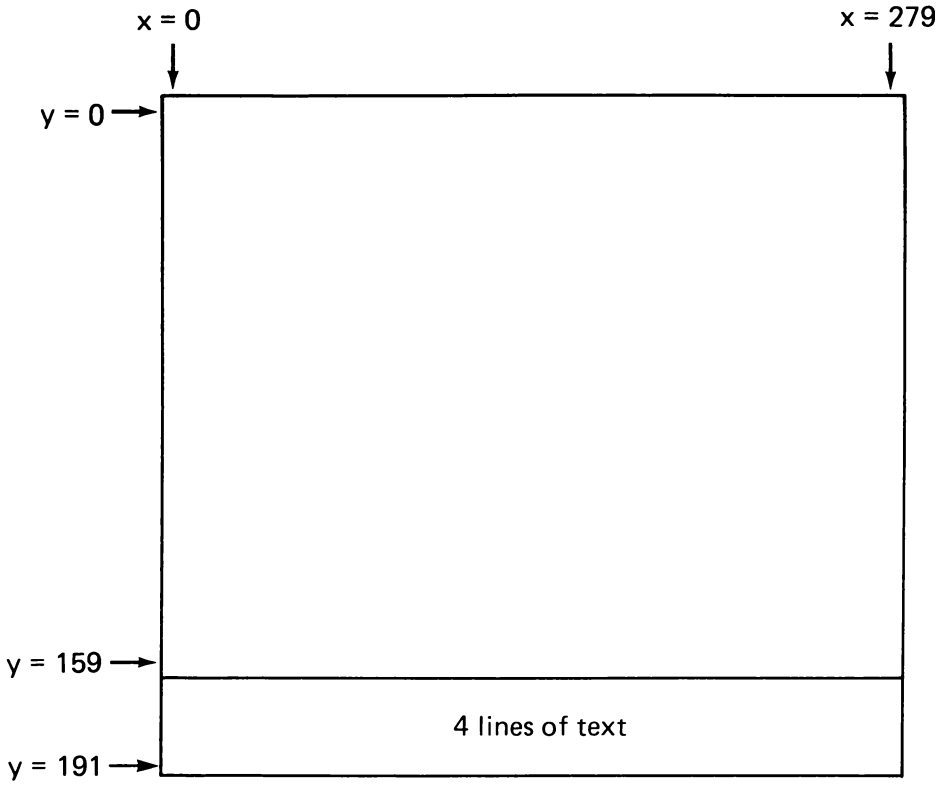


FIGURE 11.2. The high-resolution graphics screen.

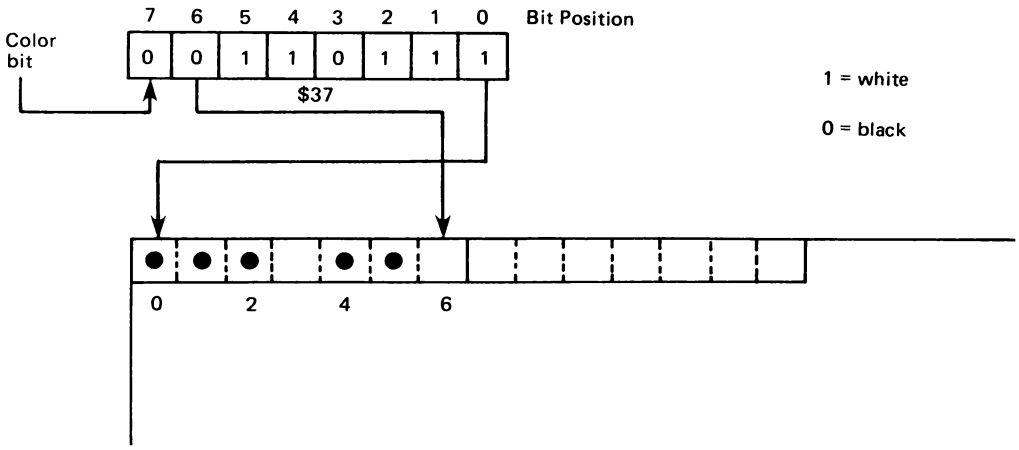


FIGURE 11.3. Relationship between data stored in the high resolution memory buffer and the image on the screen.

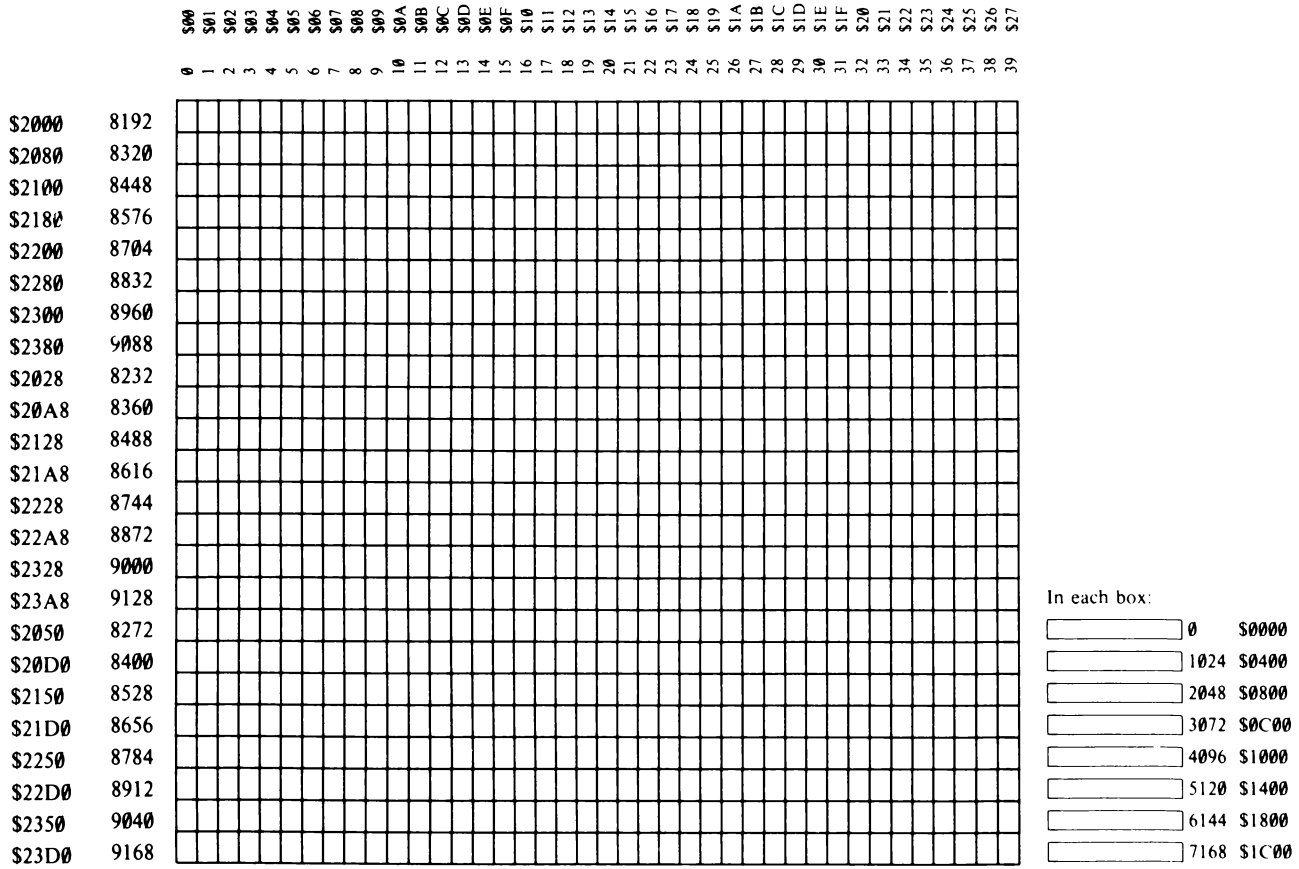
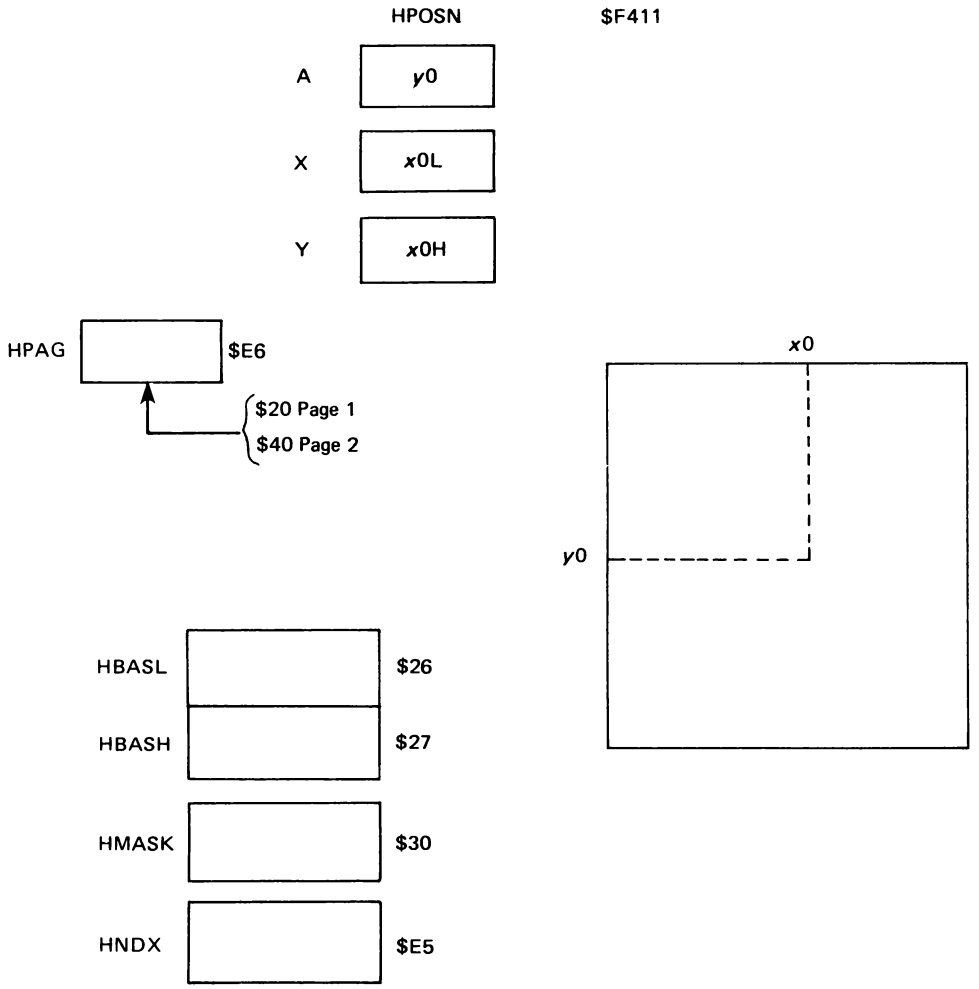


FIGURE 11.4. Memory map of high-resolution graphics screen.



Example:

```
0300  A9  20          LDA #$20
0302  85  E6          STA HPAG
0304  A9  00          LDA #$00
0306  A8              TAX
0307  A2  25          LDX #$25
0309  20  11  F4      JSR  HPOSN
030C  00              BRK
```

FIGURE 11.5. HPOSN calculates the address of the screen coordinates *x*0,*y*0.

calling HPOSN, the base address $2000 (first address in the row $y0$), is stored in HBASL ($26) and HBASH ($27). The number of bytes that must be added to this base address to give the address of the point $x0,y0$ is stored in HNDX ($E5). For $x0 = 37$, this value is 5, since seven dot positions are used in each byte. The third bit position from the right in memory location $2005 is the bit associated with the screen coordinates $x0,y0$. This bit position within the byte is specified by the bit (0–6) that is set in HMASK ($30). In the example in Figure 11.5 the value of HMASK is $84 or 10000100 which has the third bit position from the right set.

If the value in memory location $E6, called HPAG, is $20 when HPOSN is called, the address calculated will be for page 1 ($2000–$3FFF). If the value in $E6 is $40, HPOSN will calculate an address in the page 2 range $4000–$5FFF.

EXERCISE 11.1

Find the address at which the image data at $x0 = 150$, $y0 = 30$ on page 1 of high-resolution graphics are stored.

CLEARING THE HIGH-RESOLUTION GRAPHICS SCREEN

It should be clear from the previous section that to clear page 1 of the high-resolution graphics screen to all black it is necessary to store $00 in memory locations $2000–$3FFF. The built-in routine HCLR at $F3F2 will clear page 1 ($2000–$3FFF) if $20 is stored in HPAG ($E6), and will clear page 2 if $40 is stored in location $E6.

The built-in routine HGR at $F3E2 will switch to the high-resolution, mixed-text graphics mode and then clear page 1 of memory. The built-in routine HGR2 at $F3DE will switch to the high-resolution, mixed-text graphics mode and then clear page 2 of memory. The routines HGR and HGR2 do not affect the page 1/page 2 switch ($C054 or $C055) shown in Figure 10.2. Thus, for example, it is possible to clear page 2 while displaying page 1. In fact, this will happen if you call HGR2 from the TUTOR monitor, which has the page 1 switch ($C054) set. The program shown in Figure 11.6 illustrates this.

```
0300   20  DE  F3  JSR HGR2    ;set hi-res graphics and clear page 2
0303   20  56  80  JSR KEYIN   ;wait for key
0306   2C  55  C0  BIT $C055   ;switch to page 2
0309   20  56  80  JSR KEYIN   ;wait for key
030C   4C  00  80  JMP MONIT   ;jump to TUTOR
```

FIGURE 11.6. HGR2 clears page 2 but soft switch $C055 must be set to display page 2.

Type in this program and run it. Note that page 1, which has not been cleared, is displayed initially. Pressing any key will switch to page 2, which has been cleared. Pressing any key will return to the TUTOR monitor.

The dot pattern shown in Figure 11.3 can be displayed at the upper-left-hand corner of the screen on page 1 by executing the program shown in Figure 11.7. Type in this program and execute it. Pressing any key will return to the TUTOR monitor.

```
0300   20 E2 F3    JSR HGR      ;set hi-res graphics and clear page 1
0303   A9 37       LDA #$37     ;store 00110111
0305   8D 00 20    STA $2000    ;at $2000
0308   20 56 80    JSR KEYIN    ;wait for key
030B   4C 00 80    JMP MONIT    ;jump to TUTOR
```

FIGURE 11.7. Program to display dot pattern shown in Figure 11.3.

HIGH-RESOLUTION COLORS

The leftmost bit in each byte of the high-resolution graphics memory buffer is a color bit that determines the color of the remaining 7 bits in the byte. If two adjacent bits in the byte are "on," the spots on the screen will be displayed as white. If the color bit (bit 7) is 1, then a 1 in bits 0, 2, 4, and 6 of the same byte will appear *blue* on the screen, while a 1 in bits 1, 3, and 5 will appear *red.* On the other hand, if the color bit is 0, then a 1 in bits 0, 2, 4, and 6 of the same byte will appear *violet,* while a 1 in bits 1, 3, and 5 will appear *green.* These results are summarized in Figure 11.8. Note that all dots associated with a particular memory location must be the same color. Only violet and blue dots can be plotted in even screen columns and only green and red dots can be plotted in odd columns. Two dots in adjacent columns will appear white.

	Color bit	
	0	*1*
Even columns	*violet*	*blue*
Odd columns	*green*	*red*
Two dots side by side appear white		

FIGURE 11.8. Screen colors are limited in high-resolution graphics.

To see these different colors, change the value $37 in location $304 in Figure 11.7 to the values shown in Figure 11.9, and rerun the program. For example, $55 should produce four violet dots, and $AA should produce three red dots. Note that $7F and $FF both produce seven white dots.

The built-in routine BKGND at $F3F6 will fill the high-resolution memory buffer—page 1 or page 2 depending upon the contents of HPAG ($E6)—with the color data in HCOLOR1. To produce constant

colors use the hex values shown in Figure 11.9. The program shown in Figure 11.10 will fill the screen with the color value in location $0304. The value of $55 should fill the screen with violet dots. Note, however, that only every other plot position in each row on the screen contains a dot. Change the value $55 in location $304 to the other values given in Figure 11.9.

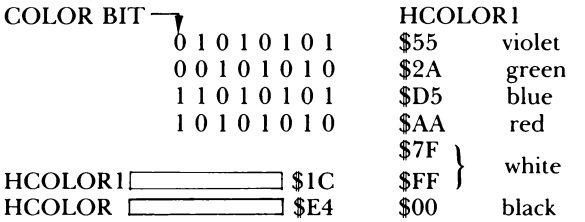


FIGURE 11.9. Hex values associated with high-resolution colors.

```
0300   20 E2 F3    JSR HGR          ;hi-res graphics
0303   A9 55       LDA #$55         ;store color data
0305   85 1C       STA HCOLOR1      ;in HCOLOR1
0307   20 F6 F3    JSR BKGND        ;fill screen
030A   20 56 80    JSR KEYIN        ;wait for key
030D   4C 00 80    JMP MONIT        ;jump to TUTOR
```

FIGURE 11.10. Program to fill the screen with a particular color.

Try storing other values in HCOLOR1 and running the program in Figure 11.10. Can you explain what you observe?

PLOTTING A SPOT

As shown in Figure 11.2, the high-resolution graphics screen is divided into a 280×192 grid (or 280×160 in the mixed-text mode). The *x*-coordinate can have values between $00 and $117 (0–279). The *y*-coordinate can have values between $00 and $BF (0–191). Before you can plot a point you must store a color data value in HCOLOR (location $E4). This can be one of the values in Figure 11.9. Then store the *y*-coordinate in accumulator A, the *x*-coordinate low in index register X, and the *x*-coordinate high in index register Y and call the built-in routine HPLOT at $F457. If you try to plot a green or red dot in an even column, or a violet or blue dot in an odd column, nothing will appear on the screen. The routine HPLOT is summarized in Figure 11.11.

The program shown in Figure 11.12 will plot a single dot near the center of the screen. Modify this program to plot the dot at different locations.

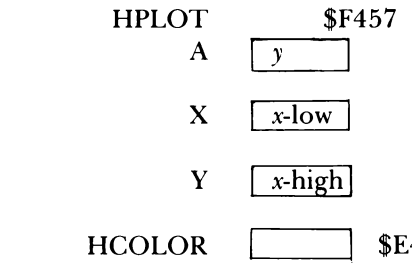


FIGURE 11.11. The subroutine HPLOT at $F457 will plot a point of color HCOLOR at *x,y*.

```
0300  20 E2 F3   JSR HGR        ;hi-res graphics
0303  A9 FF      LDA #$FF       ;white color
0305  85 E4      STA HCOLOR
0307  A9 50      LDA #$50       ;y = 80
0309  A2 8C      LDX #$8C       ;x = 140
030B  A0 00      LDY #$00
030D  20 57 F4   JSR HPLOT      ;plot point
0310  20 56 80   JSR KEYIN      ;wait for key
0313  4C 00 80   JMP MONIT      ;jump to TUTOR
```

FIGURE 11.12. Program to plot a point at $x = 140$, $y = 80$.

PLOTTING A LINE

The built-in routine HLIN at $F53A will plot a line from the most recently plotted point (using HPLOT or HLIN) to the point *x,y*. Before calling HLIN, store *y* in index register Y, *x*-low in accumulator A, and *x*-high in index register X, as shown in Figure 11.13. Note that these are not the same registers used for *x,y* in HPLOT.

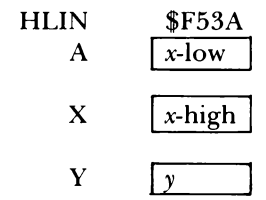


FIGURE 11.13. The subroutine HLIN at $F53A will plot a line from the most recently plotted point to *x,y*.

As an example of using HLIN, *add* the statements shown in Figure 11.14 to the program in Figure 11.12 (the last statement in Figure 11.12 is replaced). When you run this program, a point will be plotted at (140,80). Pressing any key will cause a line to be drawn from this point to (32,144). Pressing any key again will cause a second line to be plotted from the end of the first line to (112,16). Pressing any key again will return to the TUTOR monitor.

```
0313   A9 20      LDA #$20     ;x= 32
0315   A2 00      LDX #$00
0317   A0 90      LDY #$90     ;y= 144
0319   20 3A F5   JSR HLIN     ;plot line
031C   20 56 80   JSR KEYIN    ;wait for key
031F   A9 70      LDA #$70     ;x= 112
0321   A2 00      LDX #$00
0323   A0 10      LDY #$10     ;y= 16
0325   20 3A F5   JSR HLIN     ;plot line
0328   20 56 80   JSR KEYIN    ;wait for key
032B   4C 00 80   JMP MONIT    ;jump to TUTOR
```

FIGURE 11.14. Add this code to the program in Figure 11.12 to plot two lines from (140,80) to (32,144) to (112,16).

SHAPE TABLES

It is possible to predefine the shape of a graphic figure, store this shape in a *shape table*, and then plot the shape using the built-in subroutines DRAW and XDRAW. The graphic figure is defined in terms of basic "move only" or "plot and move" operations. Each "move" is one screen unit up, down, left, or right. The numbers 0–7 are used to define the eight basic move operations shown in Figure 11.15. A particular graphic figure is defined by simply listing a sequence of numbers (0–7). For exam-

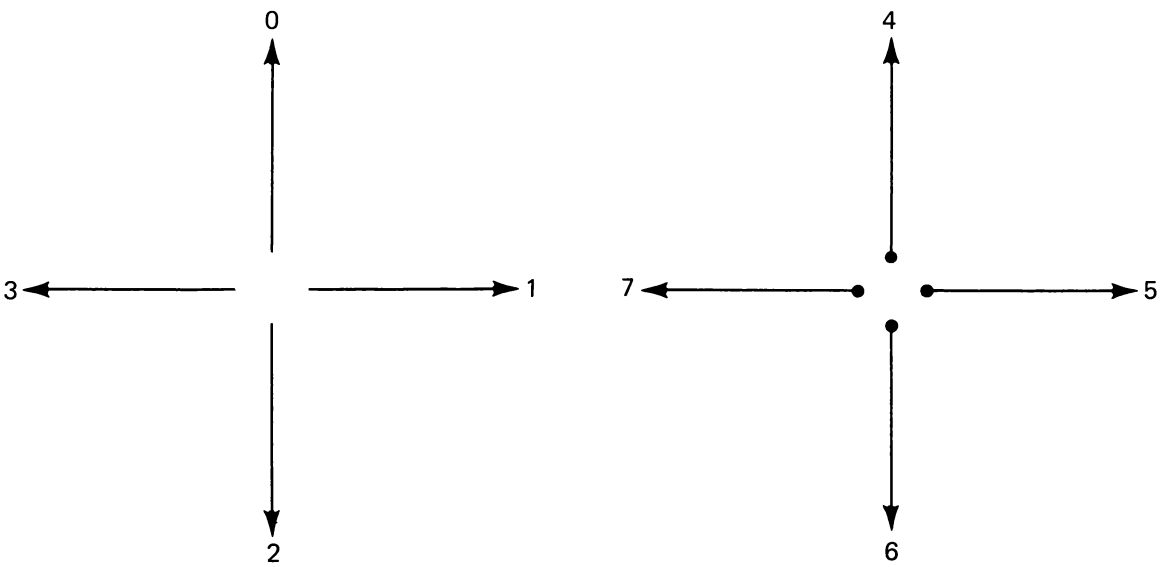


FIGURE 11.15. Numbers 0–3 define the four "move only" operations and numbers 4–7 define the four "plot and move" operations.

ple, the cross shown in Figure 11.16 can be drawn by starting at the center of the cross and carrying out the following operations:

2, 2, 2, 7, 4, 4, 7, 7, 4, 4, 5, 5, 4, 4,
5, 5, 6, 6, 5, 5, 6, 6, 7, 7, 6, 6, 7

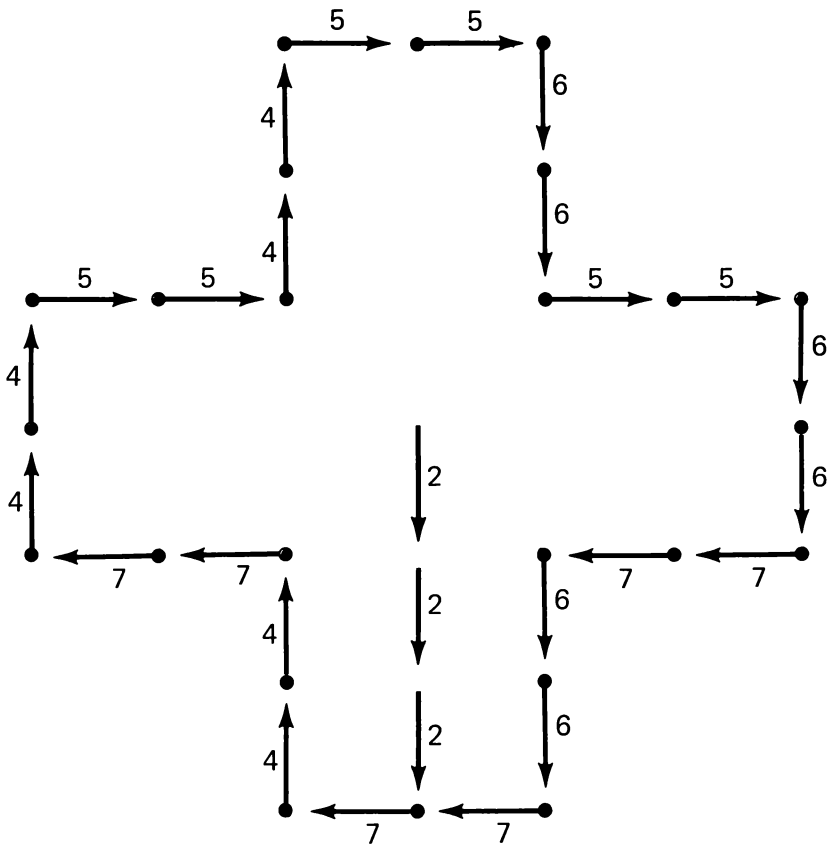


FIGURE 11.16. Defining a shape in terms of basic "move only" and "plot and move" operations.

Once a shape is defined in terms of a sequence of numbers, these numbers must be stored in the computer memory in the form of a *shape table.* After the shape table is stored in the computer memory you can draw the shape anywhere on the screen by using the DRAW and XDRAW subroutines. These subroutines allow the shape to be drawn with different sizes and orientations.

Each of the basic move operations is described by a number between 0 and 7. This is a 3-bit number. We can therefore store two of these numbers in a byte of memory. To store a shape table, list the sample numbers in pairs, starting right to left, as shown in Figure 11.17. Write each of these shape numbers as a *3-bit* binary number with two leading 0s as

shown in the center column of Figure 11.17.* Next write the binary numbers in the center column as hexadecimal values, as shown on the right in Figure 11.17. This is the shape table. It always ends with the value $00.

You can load this shape table anywhere in memory. Load it in memory starting at location $300. It will end at location $30E.

After you have stored the shape table, you can plot the shape by calling the built-in routine DRAW at $F601. You must previously have called HPOSN to define the coordinates *x*0,*y*0 where you want the shape to be drawn. Before calling DRAW you must store the starting address of the shape table in index registers X (low byte) and Y (high byte). In addition, you must store the rotation factor in accumulator A and the scaling factor in location $E7. These procedures are summarized in Figure 11.18.

	start here				
2	2	00	010	010	$12
7	2	00	111	010	$3A
4	4	00	100	100	$24
7	7	00	111	111	$3F
4	4	00	100	100	$24
5	5	00	101	101	$2D
4	4	00	100	100	$24
5	5	00	101	101	$2D
6	6	00	110	110	$36
5	5	00	101	101	$2D
6	6	00	110	110	$36
7	7	00	111	111	$3F
6	6	00	110	110	$36
0	7 ← last shape number	00	000	111	$07
00		00	000	000	$00 ← end of shape table

FIGURE 11.17. Forming a shape table for the cross in Figure 11.16.

* The Apple II will actually allow the two leading 0s to be replaced with the 1, 2, or 3 move in Figure 11.15; thus sometimes you can have three moves per byte. Since it doesn't often happen that these moves occur in exactly this location, we will always use only two basic moves per byte.

As an example, the program shown in Figure 11.19 will draw the cross defined by the shape table in Figure 11.17. Remember that you must have entered this shape table starting at location $300. The cross will be drawn centered at (140,80) because of the input values used when calling HPOSN. A scale factor of 10 is defined at location $321. This means that each defined move will be 10 units long. A rotation factor of 0 is defined at location $325. Key in this program and run it. The result is shown in Figure 11.20. Change the rotation factor in location $325 to $08 and rerun the program. The result should be as shown in Figure 11.21. Try running the program with different scale factors and different rotation factors. A scale factor, (SCALE), of 1 will cause the shape to be plotted at the defined size.

```
DRAW        $F601
XDRAW       $F65D
    A—rotation factor
    X—shape table low
    Y—shape table high
  $E7—scaling factor
Uses locations:    $F9—rotation factor
                   $E8—shape table low
                   $E9—shape table high
```

FIGURE 11.18. DRAW will draw the shape stored in the shape table at the screen position previously determined by HPOSN.

```
0310   20 E2 F3   JSR HGR        ;hi-res graphics
0313   A9 FF      LDA #$FF       ;white color
0315   85 E4      STA HCOLOR
0317   A9 50      LDA #$50       ;y0 = 80
0319   A2 8C      LDX #$8C       ;x0 = 140
031B   A0 00      LDY #$00
031D   20 11 F4   JSR HPOSN      ;calc. screen address
0320   A9 0A      LDA #$0A
0322   85 E7      STA $E7        ;scale factor = 10
0324   A9 00      LDA #$00       ;rotation factor = 0
0326   A2 00      LDX #$00       ;shape table
0328   A0 03      LDY #$03       ;at $300
032A   20 01 F6   JSR DRAW       ;draw cross
032D   20 56 80   JSR KEYIN      ;wait for key
0330   4C 00 80   JMP MONIT      ;jump to TUTOR
```

FIGURE 11.19. Program to plot cross using shape table in Figure 11.17 that must have been stored starting at location $300.

A rotation factor (ROT), of 0 will cause the shape to be plotted at the defined orientation. The value of ROT must be between 0 and $FF. Values of $10, $20, and $30 will cause rotations of 90, 180, and 270 degrees clockwise, as shown in Figure 11.22. The number of different values of

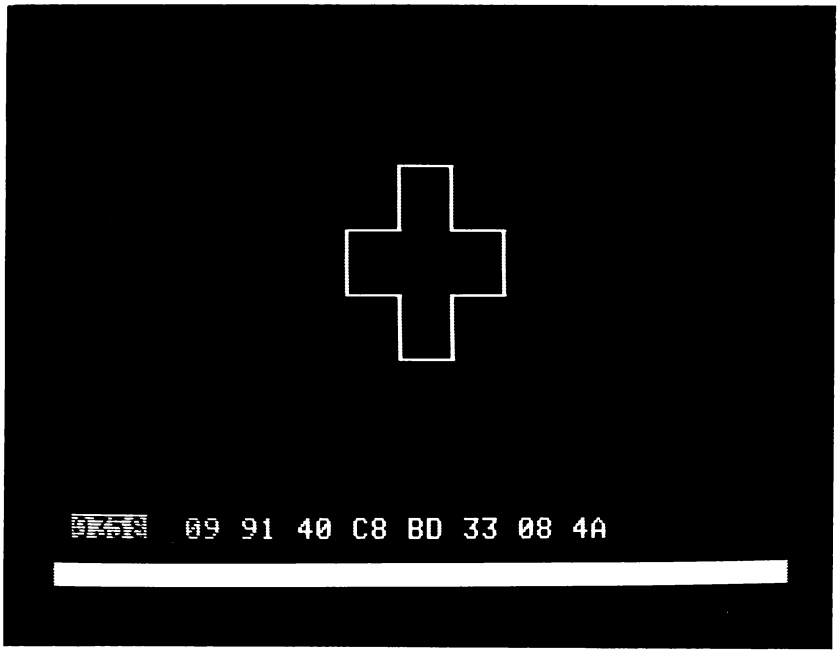


FIGURE 11.20. Result of running the program in Figure 11.19.

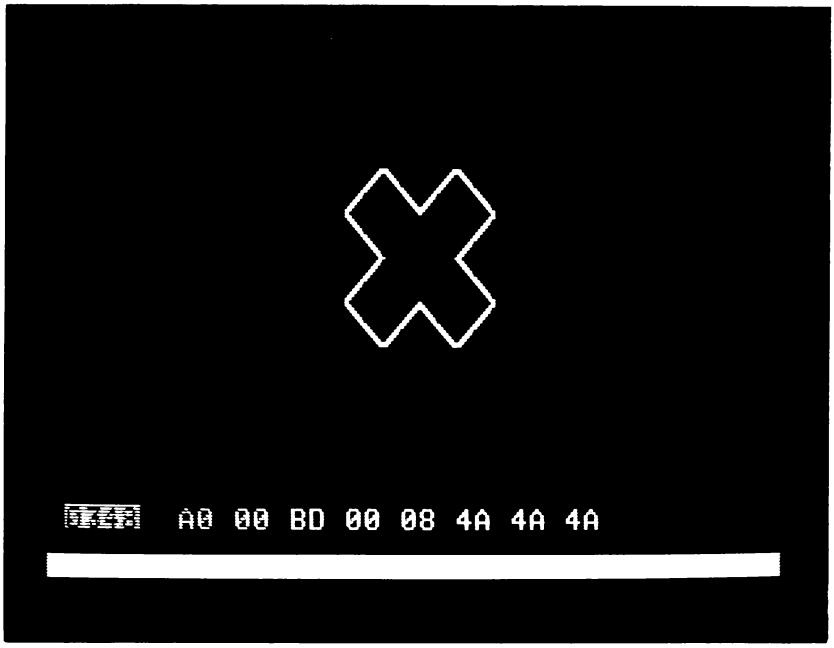


FIGURE 11.21. Result of running the program in Figure 11.19 with a rotation factor of $08.

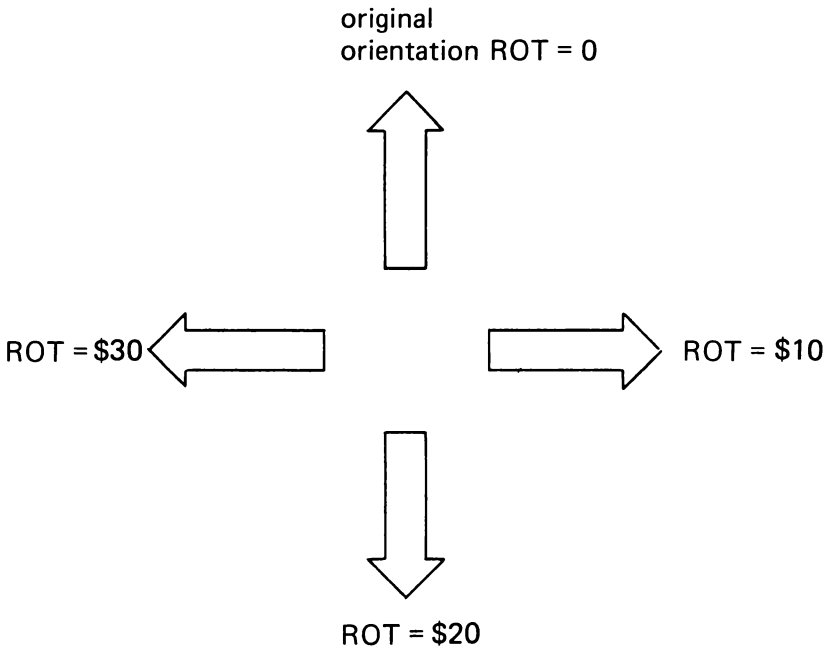


FIGURE 11.22. Increasing ROT by $10 will cause a 90-degree rotation clockwise.

ROT that are recognized depends on the value of SCALE. For a value of SCALE = 1, only the four values 0, $10, $20, and $30 shown in Figure 11.22 are recognized. Other values will normally be treated as the next smaller recognized value. For example, values of ROT between $10 and $20 will behave like $10 (90-degree rotation). Larger values of SCALE will cause more values of ROT to be recognized. For example, eight values of ROT are recognized when SCALE is equal to 2.

The subroutine XDRAW at $F65D behaves much like the DRAW statement. The only difference is that it plots the shape using the *complement* of the color that already exists at that particular location on the screen. If the existing color is white, then XDRAW will plot in black. This makes it easy to erase a figure. For example if you XDRAW a figure white (HCOLOR = $FF) and then XDRAW it again (without changing HCOLOR), it will disappear. Try it by changing location $32B in Figure 11.19 to $5D (this changes DRAW to XDRAW). Then change the program starting at location $330 to

```
0330   C9 8D       CMP #$8D     ;RETURN?
0332   D0 E3       BNE $317     ;no, go to $317
0334   4C 00 80    JMP MONIT    ;jump to TUTOR
```

This will cause the program to branch to location $317 when any key other than RETURN is pressed. Thus XDRAW will be executed at the same

location each time you press a key. The cross should continue to disappear and then reappear each time you press a key. If you press the RETURN key, you will return to the TUTOR monitor.

A summary of the high-resolution graphics routines is given in Table 11.1.

Table 11.1 High-Resolution Graphics Routines

Name	*Location*	*Arguments*	*Function*
HGR	$F3E2	none	set hi-res & clear page 1
HGR2	$F3DE	none	set hi-res & clear page 2
HCLR	$F3F2	none	clear page in HPAG
BKGND	$F3F6	none	fill screen with HCOLOR1
HPOSN	$F411	A = *y*0 X = *x*0-low Y = *x*0-high	find address of *x*0,*y*0
HPLOT	$F457	A = *y*0 X = *x*0-low Y = *x*0-high	plot point at *x*0,*y*0
HLIN	$F53A	A = *x*0-low X = *x*0-high Y = *y*0	plot line to *x*0,*y*0
DRAW	$F601	A = ROT factor X = shape table low Y = shape table high $E7 = SCALE factor	draw shape in shape table
XDRAW	$F65D	A = ROT factor X = shape table low Y = shape table high $E7 = SCALE factor	draw shape with complement color

EXERCISE 11.2

Write a program to plot a 50×50 square, using HPLOT and HLIN.

EXERCISE 11.3

Store the shape of a square in a shape table and plot a 50×50 square using the DRAW subroutine.

12

Using the Game I/O Connector

The Apple II game paddles are connected through a 16-pin socket on the motherboard. This connector can be used for a wide range of applications other than game paddles. In this chapter you will learn

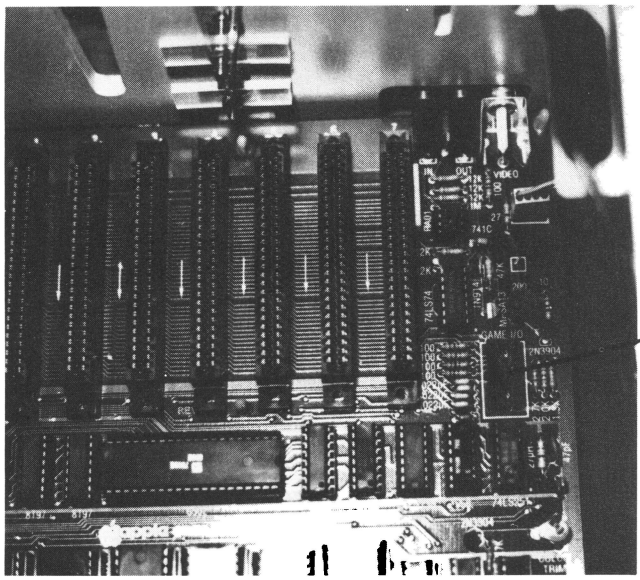
1. what each pin on the game I/O socket does
2. how the game paddles are read using the analog inputs
3. how the pushbutton inputs are used
4. how a six-channel A/D converter can be connected to the game I/O socket using the annunciator outputs and a TTL (pushbutton) input

THE GAME I/O CONNECTOR

The game I/O connector is the 16-pin socket shown in Figure 12.1. Pin 5 is a strobe output that goes low for half of a clock cycle any time an address in the range $C040–$C04F is accessed. Pins 12–15 are four TTL outputs called annunciator outputs. Accessing the addresses given in Figure 12.2 will cause these outputs to go low (OFF) or high (ON). For example, the statement BIT $C05B will cause AN1 (pin 14) to go high, while the statement BIT $C05E will cause AN3 (pin 12) to go low.

Pins 2–4 in Figure 12.1 are three TTL (pushbutton) inputs called PB0–PB2. These are single-bit inputs connected to data bit 7 of the data

FIGURE 12.1.
The game I/O connector.



+5v	1 ○	16	NC
PBØ	2	15	ANØ
PB1	3	14	AN1
PB2	4	13	AN2
$\overline{\text{CØ4Ø}}$ $\overline{\text{STROBE}}$	5	12	AN3
GCØ	6	11	GC3
GC2	7	10	GC1
Gnd	8	9	NC

ANN	*Pin*	*OFF (0 Volts)*	*ON (5 Volts)*
0	15	$C058	$C059
1	14	$C05A	$C05B
2	13	$C05C	$C05D
3	12	$C05E	$C05F

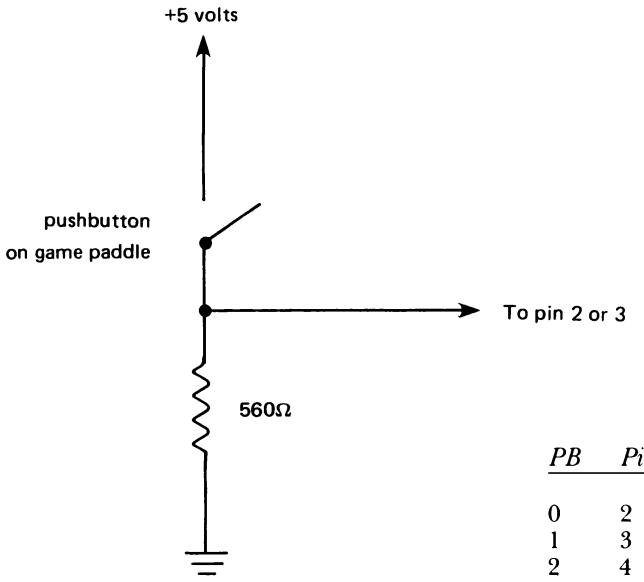
FIGURE 12.2. Addresses associated with the four annunciator (TTL) outputs.

bus. They are accessed using the addresses given in Figure 12.3. The pushbutton on game paddle 0 will cause bit 7 of location $C061 to be 1 while the pushbutton is being pressed. Similarly, bit 7 of location $C062 will be 1 while the pushbutton on game paddle 1 is being pressed (See Figure 12.3).

The game paddles themselves are read using the game controller inputs GC0–GC3 (pins 6–7, 10–11). These inputs will be described in the next section.

ANALOG GAME CONTROLLER INPUTS

The four inputs GC0–CG3 shown in Figure 12.1 are connected to a 558 quad timer chip. If these input pins are connected to +5 volts through 150kΩ potentiometers, the quad timer behaves like four monostable



PB	*Pin*	*Address*
0	2	$C061
1	3	$C062
2	4	$C063

FIGURE 12.3. Addresses associated with the three pushbutton (TTL) inputs.

multivibrators (one shots) that are triggered by the address $C070. When this happens, the four monostable outputs of the 558 quad timer go high. These outputs correspond to bit 7 of the addresses $C064–$C067, as shown in Figure 12.4. These monostable outputs (bit 7) will remain high for a time τ that depends on the setting of the four potentiometers connected to the game controller inputs. The game paddles use only the two inputs GC0 and GC1.

$C070 triggers monostable

CG	*Pin*	*Address*
0	6	$C064
1	10	$C065
2	7	$C066
3	11	$C067

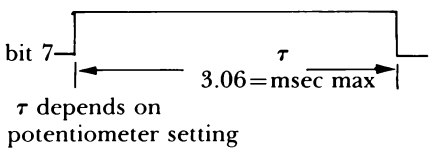


FIGURE 12.4. Addresses associated with the four analog (game controller) inputs.

To obtain a value proportional to a potentiometer setting, it is necessary to increment a register during the time that the monostable output is high. This is done in the built-in routine PREAD at location $FB1E. This routine, shown in Figure 12.5, expects the value of the game controller input used, GC (0–3), to be in index register X when the routine is called. It returns a value between $00 and $FF in index register Y. This value is

proportional to the value of the potentiometer setting connected to the particular GC input.

Caution must be exercised when using the routine PREAD. Each time it is called it triggers *all* GC monostables. But the monostable output must have come low from a previous trigger. This means that if you try to read the values from two game paddles, you may get erroneous results unless you delay between the two calls to PREAD to allow time for all monostables to return to 0.

```
PREAD    LDA   $C070     ;Trigger paddles
         LDY   #$00      ;Initialize count
         NOP             ;Compensate for
         NOP             ;first count
PREAD2   LDA   $C064,X
         BPL   RTS2D     ;Count Y register
         INY             ;every 12 μsec
         BNE   PREAD2    ;Exit at
         DEY             ;255 max.
RTS2D    RTS
```

FIGURE 12.5. Built-in routine PREAD at location $FB1E uses GC value (0–3) in index register X and returns potentiometer value ($00–$FF) in index register Y.

The program shown in Figure 12.6 on the following page will read an X value equal to the reading of paddle 0 divided by 8. It will then read (after an appropriate delay) a Y value equal to the reading of paddle 1 divided by 8. A low-resolution spot is then plotted at X,Y and the process is continued until you press any key on the keyboard.

Type in this program and run it. Try reducing the delay time between PREAD calls by changing the value in location $33D which controls the time delay produced by the subroutine DELAY. Can you explain the result when you rerun the program?

EXERCISE 12.1

Modify the program in Figure 12.6 so that the color plotted is changed (by incrementing the value in COLOR) each time the pushbutton on paddle 0 is pressed. Have the program clear the screen each time the pushbutton on paddle 1 is pressed.

SIX-CHANNEL A/D CONVERTER

The Motorola MC14443 is an analog-to-digital-converter linear subsystem whose operation is described in the data sheet given in Appendix D. This chip uses three inputs (A0–A2) to select one of eight input channels whose voltage is to be measured. These input lines will be connected to three annunciator outputs of the game I/O connector. The ramp start input will be connected to the fourth annunciator output.

```
                        0010  ;       GAME PADDLES
                        0020  MONIT   EQU 8000
                        0030  PLOT    EQU F800
                        0040  SETCOL  EQU F864
                        0050  PREAD   EQU FB1E
                        0060  SETGR   EQU FB40
                        0070  COLOR   EQU 300
                        0080          ORG 308
0308  2040FB    0090  PADDLE  JSR SETGR
030B  A901      0100          LDA #$01
030D  8D0003    0110          STA COLOR
0310  A200      0120  LOOP    LDX #$00
0312  201EFB    0130          JSR PREAD
0315  98        0140          TYA
0316  4A        0150          LSR
0317  4A        0160          LSR
0318  4A        0170          LSR
0319  48        0180          PHA
031A  203C03    0190          JSR DELAY
031D  A201      0200          LDX #$01
031F  201EFB    0210          JSR PREAD
0322  98        0220          TYA
0323  4A        0230          LSR
0324  4A        0240          LSR
0325  4A        0250          LSR
0326  A8        0260          TAY
0327  AD0003    0270          LDA COLOR
032A  2064F8    0280          JSR SETCOL
032D  68        0290          PLA
032E  2000F8    0300          JSR PLOT
0331  203C03    0310          JSR DELAY
0334  2C00C0    0320          BIT $C000
0337  10D7      0330          BPL LOOP
0339  4C0080    0340          JMP MONIT
                        0350  ;
                        0360  ;       DELAY
033C  A200      0370  DELAY   LDX #$00
033E  CA        0380  DLY1    DEX
033F  EA        0390          NOP
0340  EA        0400          NOP
0341  EA        0410          NOP
0342  EA        0420          NOP
0343  D0F9      0430          BNE DLY1
0345  60        0440          RTS
```

FIGURE 12.6. Program to plot spots on the screen corresponding to the game paddle settings.

When the ramp start signal goes high the capacitor, which has been charged to the input voltage, is discharged toward 0 volts. When the capacitor voltage reaches the comparator threshold voltage, the comparator output goes low. This output is connected to the pushbutton (TTL) input PB0. If an index register is incremented during the time the capacitor is being discharged, the resulting value in the index register will be proportional to the input voltage being measured.

The program shown in Figure 12.7 will continually display the hex value of the voltage on channel 1 on the upper-left-hand corner of the

```
                        0010  ;        A/D CONVERTER
                        0020  CH       EQU 24
                        0030  BASCLC   EQU FBC1
                        0040  PRBYTE   EQU FDDA
                        0050  AOON     EQU C059
                        0060  A1OFF    EQU C05A
                        0070  A2OFF    EQU C05C
                        0080  RSON     EQU C05F
                        0090  RSOFF    EQU C05E
                        0100  CO       EQU C061
                        0110           ORG 300
0300   A900     0120  ADCONV   LDA #$00
0302   8524     0130           STA CH
0304   20C1FB   0140           JSR BASCLC
0307   2C59C0   0150           BIT AOON
030A   2C5AC0   0160           BIT A1OFF
030D   2C5CC0   0170           BIT A2OFF
0310   2C5EC0   0180  AGAIN    BIT RSOFF
0313   202F03   0190           JSR DELAY
0316   A201     0200           LDX #$01
0318   18       0210           CLC
0319   2C5FC0   0220           BIT RSON
031C   2C61C0   0230  LOOP     BIT CO
031F   1003     0240           BPL DONE
0321   E8       0250           INX
0322   90F8     0260           BCC LOOP
0324   8A       0270  DONE     TXA
0325   20DAFD   0280           JSR PRBYTE
0328   C624     0290           DEC CH
032A   C624     0300           DEC CH
032C   4C1003   0310           JMP AGAIN
                        0320  ;
                        0330  ;        DELAY
032F   A230     0340  DELAY    LDX #$30
0331   CA       0350  DLY1     DEX
0332   D0FD     0360           BNE DLY1
0334   60       0370           RTS
```

FIGURE 12.7. Program to display the voltage on channel 1 of the MC14443 A/D converter.

screen. Channel 1 is selected by accessing addresses $C059, $C05A, and $C05C. Each time the ramp start is brought low ($C05E), a delay gives the capacitor time to charge to the input voltage. The ramp start is then brought high ($C05F) and index register X is incremented until the comparator output ($C061) goes low.

The circuit for connecting the A/D converter to the game I/O plug is shown in Figure 12.8. The resistor and capacitor values can be determined as follows. From the data sheet (see Appendix D) the reference voltage (pin 8) should be between V_{SS} + 2 V (2 volts) and V_{DD} − 2 V (3 volts). Make it 2.5 volts by setting $R_1 = R_2 = 1$ KΩ. The reference current range I_R should be between 10 and 50 μAdc. Pick 30μA and therefore choose R_{Ref} to be

$$R_{Ref} = \frac{5\text{ V}}{30\ \mu\text{A}} = 167\text{ K}$$

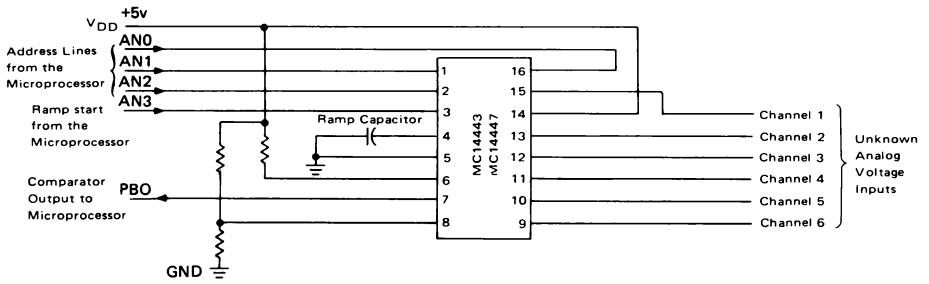


FIGURE 12.8. Connecting the MC14443 A/D converter to the game I/O plug.

Select the capacitor C so that the time t_R to discharge the reference voltage is less than 255 times the time to execute the LOOP in Figure 12.7. This will ensure that the maximum displayed value is less than FF. It takes about 12 μsec to go around this loop once, or about 3 msec to go around 250 times. The slope of the voltage discharge curve is I_R/C (see data sheet) so that

$$\frac{I_R}{C} = \frac{2.5\text{ V}}{3\text{ msec}}$$

from which

$$C = \frac{3 \times 10^{-3} \times 30 \times 10^{-6}}{2.5} = 36 \times 10^{-9} = .036\ \mu\text{F}$$

13

Using the Peripheral I/O Slots

The game I/O socket described in Chapter 12 is convenient to use but has limited capabilities. When more extensive interfacing is required you can use the peripheral I/O slots on the Apple II motherboard. There are eight of these peripheral slots, as shown in Figure 13.1. In this chapter you will learn

1. what signals are available at the pins of the peripheral I/O connector
2. how the Apple II decodes I/O addresses and assigns them to particular peripheral slots
3. how a PROM on the peripheral I/O board can be used to make a smart interface board

THE PERIPHERAL I/O CONNECTOR

The pinout for each of the peripheral connectors is shown in Figure 13.2. The description of each signal is given in Table 13.1. Note that all the address lines and data lines as well as a variety of control signals are brought out to each peripheral connector. The pins labeled $\overline{\text{I/O SELECT}}$, $\overline{\text{DEVICE SELECT}}$, and $\overline{\text{I/O STROBE}}$ are used for selecting various memory locations on the peripheral board. These memory addresses are all in the I/O memory range $C000–$CFFF.

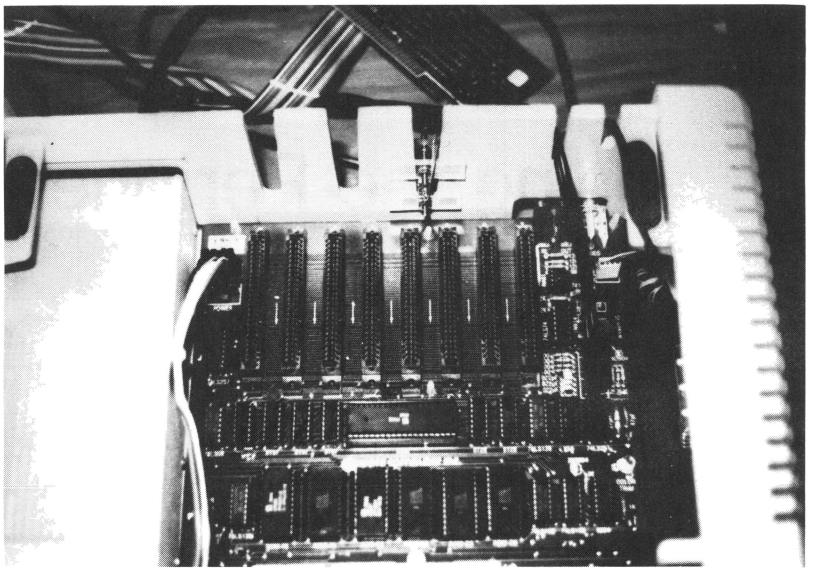


FIGURE 13.1. Eight peripheral I/O slots are located on the Apple II motherboard.

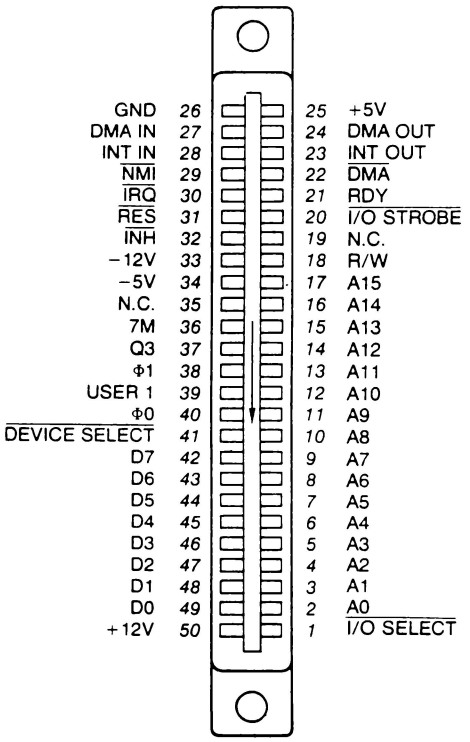


FIGURE 13.2.
Peripheral connector pinout.

Table 13.1 Peripheral Connector Signal Description

Pin:	Name:	Description:
1	$\overline{\text{I/O SELECT}}$	This line, normally high, will become low when the microprocessor references page $C*n*, where *n* is the individual slot number. This signal becomes active during Φ0 and will drive 10 LSTTL loads*. This signal is not present on peripheral connector 0.
2-17	A0-A15	The buffered address bus. The address on these lines becomes valid during Φ1 and remains valid through Φ0. These lines will each drive 5 LSTTL loads*.
18	R/$\overline{\text{W}}$	Buffered Read/$\overline{\text{Write}}$ signal. This becomes valid at the same time the address bus does, and goes high during a read cycle and low during a write. This line can drive up to 2 LSTTL loads*.
19	SYNC	On peripheral connector 7 *only*, this pin is connected to the video timing generator's SYNC signal.
20	$\overline{\text{I/O STROBE}}$	This line goes low during Φ0 when the address bus contains an address between $C800 and $CFFF. This line will drive 4 LSTTL loads*.
21	RDY	The 6502's RDY input. Pulling this line low during Φ1 will halt the microprocessor, with the address bus holding the address of the current location being fetched.
22	$\overline{\text{DMA}}$	Pulling this line low disables the 6502's address bus and halts the microprocessor. This line is held high by a 3KΩ resistor to +5v.
23	INT OUT	Daisy-chained interrupt output to lower priority devices. This pin is usually connected to pin 28 (INT IN).
24	DMA OUT	Daisy-chained DMA output to lower priority devices. This pin is usually connected to pin 22 (DMA IN).
25	+5v	+5 volt power supply. 500mA current is available for *all* peripheral cards.
26	GND	System electrical ground.

*Loading limits are for each peripheral card.

(continued)

Table 13.1 (*continued*)

Pin:	Name:	Description:
27	DMA IN	Daisy-chained DMA input from higher priority devices. Usually connected to pin 24 (DMA OUT).
26	INT IN	Daisy-chained interrupt input from higher priority devices. Usually connected to pin 23 (INT OUT).
29	$\overline{\text{NMI}}$	Non-Maskable Interrupt. When this line is pulled low the Apple begins an interrupt cycle and jumps to the interrupt handling routine at location $3FB.
30	$\overline{\text{IRQ}}$	Interrupt ReQuest. When this line is pulled low the Apple begins an interrupt cycle only if the 65Ø2's I (Interrupt disable) flag is not set. If so, the 65Ø2 will jump to the interrupt handling subroutine whose address is stored in locations $3FE and $3FF.
31	$\overline{\text{RES}}$	When this line is pulled low the microprocessor begins a RESET cycle (see page 36).
32	$\overline{\text{INH}}$	When this line is pulled low, all ROMs on the Apple board are disabled. This line is held high by a 3KΩ resistor to +5v.
33	−12v	−12 volt power supply. Maxmum current is 200mA for all peripheral boards.
34	−5v	−5 volt power supply. Maximum current is 200mA for all peripheral boards.
35	COLOR REF	On peripheral connector 7 *only*, this pin is connected to the 3.5MHz COLOR REFerence signal of the video generator.
36	7M	7MHz clock. This line will drive 2 LSTTL loads*.
37	Q3	2MHz asymmetrical clock. This line will drive 2 LSTTL loads*.
38	Φ1	Microprocessor's phase one clock. This line will drive 2 LSTTL loads*.
39	USER 1	This line, when pulled low, disables *all* internal I/O address decoding.

Table 13.1 (*continued*)

Pin:	Name:	Description:
40	Φ0	Microprocessor's phase zero clock. This line will drive 2 LSTTL loads*.
41	$\overline{\text{DEVICE SELECT}}$	This line becomes active (low) on each peripheral connector when the address bus is holding an address between $C0*n*0 and $C0*n*F, where *n* is the slot number plus $8. This line will drive 10 LSTTL loads*.
42-49	D0-D7	Buffered bidirectional data bus. The data on this line becomes valid 300nS into Φ0 on a write cycle, and should be stable no less than 100ns before the end of Φ0 on a read cycle. Each data line can drive one LSTTL load.
50	+12v	+12 volt power supply. This can supply up to 250mA total for all peripheral cards.

MEMORY DECODING IN THE APPLE II

Memory devices such as RAMs, ROMs, PROMs, and memory-mapped I/O devices such as PIAs (see Chapter 14) have *chip select* pins. These are labeled as CS or $\overline{\text{CS}}$. For a particular chip to be selected, all CS pins must be high (5 volts) and all $\overline{\text{CS}}$ pins must be low (0 volts). It is up to the system designer to make sure that a particular memory chip is selected only when it is being addressed. This is done by the proper decoding of the address lines.

When you design a microprocessor-based system you must assign particular addresses to each memory chip. For example, the memory map of the Apple II is given in Figure 10.1 of Chapter 10. The 48K bytes of RAM between locations $0000 and $BFFF are made up of three rows of eight chips each. Each chip in a row is a 16K×1 dynamic RAM that contributes 1 bit to the data bus. Thus, each row contains 16K bytes of memory.

The Apple II contains six ROMs that occupy the memory range $D000–$FFFF. Each ROM contains 2K bytes of memory. The remaining 4K bytes of memory between $C000 and $CFFF are used for I/O.

The first step in determining how to decode this memory space is to fill in a system layout work sheet. Figure 13.3 shows such a layout sheet for the Apple II. In this figure a 1 means that the address line must be high to enable the device and a 0 means that the address line must be low

SYSTEM LAYOUT WORK SHEET

	MPU ADDRESS LINES A15-A0																ADDRESS	
DEVICE	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	FROM	TO
RAM(row1)	0	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	0000	3FFF
RAM(row2)	0	1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	4000	7FFF
RAM(row3)	1	0	X	X	X	X	X	X	X	X	X	X	X	X	X	X	8000	BFFF
I/O	1	1	0	0	–	–	–	–	–	–	–	–	–	–	–	–	C000	CFFF
ROM 1	1	1	0	1	0	X	X	X	X	X	X	X	X	X	X	X	D000	D7FF
ROM 2	1	1	0	1	1	X	X	X	X	X	X	X	X	X	X	X	D800	DFFF
ROM 3	1	1	1	0	0	X	X	X	X	X	X	X	X	X	X	X	E000	E7FF
ROM 4	1	1	1	0	1	X	X	X	X	X	X	X	X	X	X	X	E800	EFFF
ROM 5	1	1	1	1	0	X	X	X	X	X	X	X	X	X	X	X	F000	F7FF
ROM 6	1	1	1	1	1	X	X	X	X	X	X	X	X	X	X	X	F800	FFFF

FIGURE 13.3. System layout work sheet for the Apple II.

to enable the device. An X indicates one of the address lines used to address the locations within the chip.

Look first at the three RAM rows. Each 16K×1 RAM chip will require 14 address lines to address the 16K memory locations in the chip. These are denoted by X in Figure 13.3. Actually, the seven address lines A0–A6 and A7–A13 are multiplexed onto the same seven pins of the memory chip. The two remaining address lines A14 and A15 will determine the address range to which a particular chip will respond. If A14 and A15 are both 0, the address range will be $0000–$3FFF. If A14 is 1 and A15 is 0, the address range will be $4000–$7FFF. If A14 is 0 and A15 is 1, the address range will be $8000–$BFFF. This is shown in the first three lines in Figure 13.3.

If A14 and A15 are both 1, the address will be above $C000. Address lines A11–A13 are used to divide the 12K range $D000–$FFFF into six 2K blocks corresponding to the six Apple II ROMs. The I/O address range $C000–$CFFF is specified by having A12 and A13 equal to 0 (and A14 and A15 equal to 1).

Having specified the address range of all memory chips in a layout sheet such as Figure 13.3, it is only necessary to decode the address lines according to this work sheet. Various kinds of circuits can be used for this decoding. The Apple II uses a 74LS138 1-of-8 decoder chip whose operation is shown in Figure 13.4. This chip will decode all addresses above $C000 when A14 and A15 are both 1. Address lines A11–A13 then determine which output line Z0–Z7 goes low. Comparing Figure 13.4 with Figure 13.3, you can see that outputs Z2–Z7 can be used directly as inputs to the $\overline{CS}$ chip select pins on the six ROMs. The Apple II ROMs also have a CS chip select pin that must be high. These pins on all six chips are tied together and pulled high through a pull-up resistor. This line is also brought out to pin 32, $\overline{INH}$, on *all* peripheral I/O slots (see Figure 13.2). This means that if you pull this line low, all six ROMs will be disabled.

I/O Decoding

From Figures 13.3 and 13.4, note that the address range $C800–$CFFF causes the output Z1 of the 74LS138 to go low. This output is sent directly to pin 20, $\overline{\text{I/O STROBE}}$, on all I/O peripheral slots (see Figure 13.2). The output Z0 of the 74LS138 in Figure 13.4 goes low for addresses between $C000 and $C7FF. This output, called $\overline{\text{I/O ADDR}}$, is used as an enable input to another 74LS138, as shown in Figure 13.5. This chip uses address lines A8–A10 to decode the addresses $C000–$C7FF into eight blocks of 256 bytes each according to the layout sheet shown in Figure 13.6. The outputs Z1–Z7 of U2 in Figure 13.5 are sent to pin 1, $\overline{\text{I/O SELECT}}$, in the I/O slots 1–7, respectively. This means that the $\overline{\text{I/O SELECT}}$ pin in a peripheral I/O slot will go low when any address in the address range C*n*00–C*n*FF is referenced, where *n* is the slot number. Note

Inputs					*Outputs*							
Enable		*Select*										
E1 + E2	*E3*	*A2*	*A1*	*A0*	*Z0*	*Z1*	*Z2*	*Z3*	*Z4*	*Z5*	*Z6*	*Z7*
1	X	X	X	X	1	1	1	1	1	1	1	1
X	0	X	X	X	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	1	1	1
0	1	0	0	1	1	0	1	1	1	1	1	1
0	1	0	1	0	1	1	0	1	1	1	1	1
0	1	0	1	1	1	1	1	0	1	1	1	1
0	1	1	0	0	1	1	1	1	0	1	1	1
0	1	1	0	1	1	1	1	1	1	0	1	1
0	1	1	1	0	1	1	1	1	1	1	0	1
0	1	1	1	1	1	1	1	1	1	1	1	0

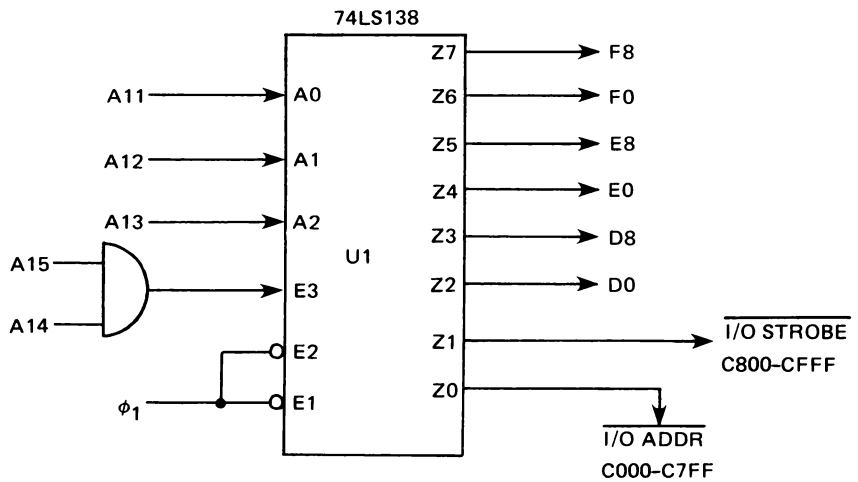


FIGURE 13.4. A 74LS138 1-of-8 decoder is used to produce chip select signals in the Apple II.

that the I/O SELECT signal goes only to slots 1–7. It is not available on slot 0.

The output Z0 of U2, which decodes $C000–$C0FF, is used as an enable input to U3, another 74LS138. It is also sent to another circuit to help decode the built-in I/O between $C000 and $C07F. The decoding chip U3 in Figure 13.5 uses address lines A4–A7 to decode the address range $C080–$C0FF into eight 16-byte blocks according to the layout work sheet shown in Figure 13.7. These outputs are sent to the eight DEVICE SELECT pins (pin 41) on the eight I/O slots. Thus, each slot has

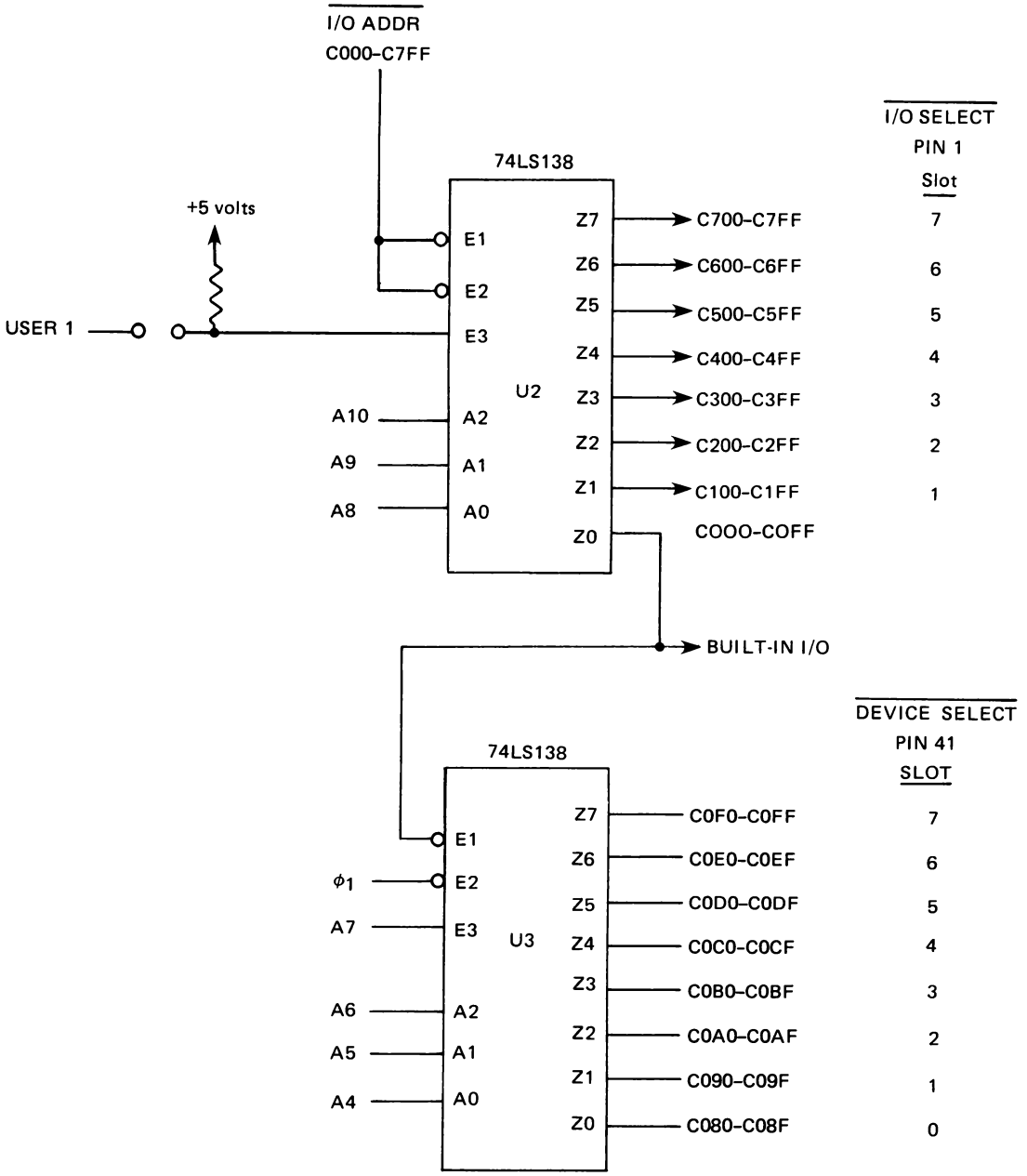


FIGURE 13.5. Peripheral I/O slot decoding.

SYSTEM LAYOUT WORK SHEET

$\overline{I/O\ SELECT}$	MPU ADDRESS LINES A15-A0																ADDRESS	
DEVICE	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	FROM	TO
SLOT 1	1	1	0	0	0	0	0	1	X	X	X	X	X	X	X	X	C100	C1FF
SLOT 2	1	1	0	0	0	0	1	0	X	X	X	X	X	X	X	X	C200	C2FF
SLOT 3	1	1	0	0	0	0	1	1	X	X	X	X	X	X	X	X	C300	C3FF
SLOT 4	1	1	0	0	0	1	0	0	X	X	X	X	X	X	X	X	C400	C4FF
SLOT 5	1	1	0	0	0	1	0	1	X	X	X	X	X	X	X	X	C500	C5FF
SLOT 6	1	1	0	0	0	1	1	0	X	X	X	X	X	X	X	X	C600	C6FF
SLOT 7	1	1	0	0	0	1	1	1	X	X	X	X	X	X	X	X	C700	C7FF
USER PROM	1	1	0	0	1	X	X	X	X	X	X	X	X	X	X	X	C800	CFFF

FIGURE 13.6. Layout work sheet for I/O SELECT addressing.

SYSTEM LAYOUT WORK SHEET

DEVICE SELECT	MPU ADDRESS LINES A15-A0																ADDRESS	
DEVICE	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	FROM	TO
BUILT-IN I/O	1	1	0	0	0	0	0	0	0	X	X	X	X	X	X	X	C000	C07F
SLOT 0	"							"	1	0	0	0					C080	C08F
SLOT 1	"							"	1	0	0	1					C090	C09F
SLOT 2	"							"	1	0	1	0					C0A0	C0AF
SLOT 3	"							"	1	0	1	1					C0B0	C0BF
SLOT 4	"							"	1	1	0	0					C0C0	C0CF
SLOT 5	"							"	1	1	0	1					C0D0	C0DF
SLOT 6	"							"	1	1	1	0					C0E0	C0EF
SLOT 7	"							"	1	1	1	1					C0F0	C0FF

FIGURE 13.7. Layout work sheet for DEVICE SELECT addressing.

16 bytes decoded for it in the address range $C0*n*0–$C0*n*F, where *n* is the slot number plus 8.

The $\overline{\text{I/O SELECT}}$ and $\overline{\text{DEVICE SELECT}}$ addresses associated with each slot are summarized in Table 13.2.

Table 13.2 I/O Peripheral Slot Addresses

Slot No.	*$\overline{\text{I/O SELECT}}$, Pin 1*	*$\overline{\text{DEVICE SELECT}}$, Pin 41*
0	No connection	$C080 –$C08F
1	$C100–$C1FF	$C090 –$C09F
2	$C200–$C2FF	$C0A0–$C0AF
3	$C300–$C3FF	$C0B0–$C0BF
4	$C400–$C4FF	$C0C0–$C0CF
5	$C500–$C5FF	$C0D0–$C0DF
6	$C600–$C6FF	$C0E0 –$C0EF
7	$C700–$C7FF	$C0F0 –$C0FF

Recall from Chapter 9 that the TV RAM uses only 960 bytes from the 1,024 bytes between $400 and $7FF. The 64 remaining bytes are reserved for use by the I/O peripheral slots as scratchpad RAM. There are eight blocks of 8 bytes each. Each slot is assigned 1 byte in each block according to Table 13.3. The base address locations are used by Apple DOS.

Table 13.3 Scratchpad RAM for I/O Slots

Base Address	
$478	Base address + $0*n*
$4F8	is available to use
$578	in slot *n* (1–7)
$5F8	
$678	
$6F8	
$778	
$7F8	

The output pin Z0 of chip U2 in Figure 13.4 goes low for addresses in the range $C000–$C0FF. This signal, called *BUILT-IN I/O*, is fed to the enable input E2 of another 74LS138, as shown in Figure 13.8. This chip decodes the built-in addresses in the range $C000–$C07F according to the layout work sheet shown in Figure 13.9. The use of these built-in I/O addresses are described elsewhere in this book.

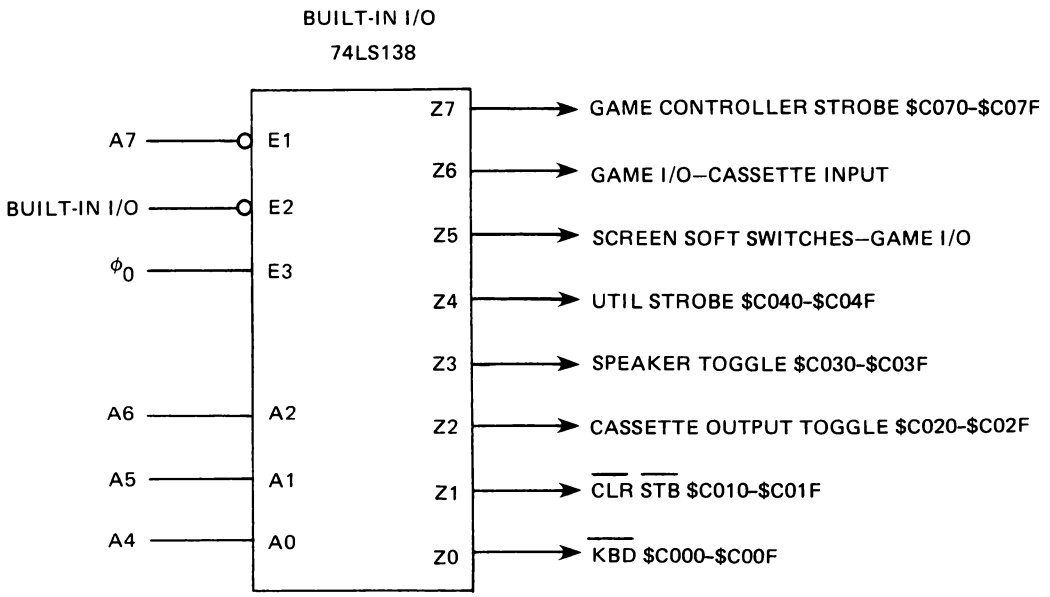


FIGURE 13.8. Decoding the built-in I/O addresses.

COMMUNICATING WITH PERIPHERAL BOARDS

A typical Apple II peripheral board will contain a 256-byte PROM that is decoded by the I/O SELECT line. The address range of this PROM will be $C*n*00–$C*n*FF, where *n* is the slot number (see Table 13.2). This PROM will contain the I/O driver routines associated with the particular peripheral board.

The memory locations $36 and $37, called CSWL (character output switch low) and CSWH (character output switch high), contain the address of the current character output routine. This is normally the address $9EBD, which is a routine in Apple DOS (the disk operating system). This routine intercepts each character going to the screen to see if it is part of a DOS command. It then sends the character to the screen by executing COUT1 (see Chapter 9).

When a BASIC program executes a PRINT statement, or when you LIST a program, the routine COUT at location $FDED is called. This routine is

```
FDED  6C 36 00  COUT  JMP (CSWL)
```

which jumps indirectly to the routine whose address is in locations $36 and $37. If you change this address in locations $36 and $37, you can cause a different character output routine to be executed.

SYSTEM LAYOUT WORK SHEET

	MPU ADDRESS LINES A15-A0																ADDRESS	
DEVICE	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	FROM	TO
KEYBOARD	1	1	0	0	0	0	0	0	0	0	0	0	X	X	X	X	C000	C00F
KBD STROBE	"							"	0	0	0	1					C010	C01F
CASSETTE OUTPUT TOGGLE	"							"	0	0	1	0					C020	C02F
SPEAKER TOGGLE	"							"	0	0	1	1					C030	C03F
UTILITY STROBE	"							"	0	1	0	0					C040	C04F
SOFT SWITCHES GAME I/O	"							"	0	1	0	1					C050	C05F
CASSETTE INPUT GAME I/O	"							"	0	1	1	0					C060	C06F
GAME CONTROL STROBE	"							"	0	1	1	1					C070	C07F

FIGURE 13.9. Layout work sheet for the built-in I/O addresses.

When the BASIC statement PR#*n* (or the monitor command *n* CTRL P) is executed (where *n* is a slot number), the address $C*n*00 is stored in locations $36 and $37 and the subroutine at $C*n*00 is executed. This is the starting address of the PROM program on the peripheral board in slot *n*. This routine will normally initialize the peripheral board, store a new entry address (between $C*n*00 and $C*n*FF) in the locations $36 and $37, and then return to BASIC (or the monitor) by executing an RTS instruction. Subsequent PRINT statements in the BASIC program will cause the routine on the peripheral board to be executed for each character printed. This routine can display the character on the screen by calling COUT1; in addition, it can send the character to some peripheral device.

When the BASIC statement

PR#0

is executed, the Apple II restores the address of $9EBD (the screen intercept entry point in DOS) to the locations $36 and $37. This means that subsequent output statements will output characters to the screen.

If you want to write a machine language program, stored in RAM, that can be used as an output driver routine for a peripheral board, you simply must store the starting address of your character output routine in locations $36 and $37. We will illustrate this procedure in the next chapter by writing a routine that will send all output characters to a printer.

Input data from a peripheral board are handled in a similar way. The INPUT statement in BASIC calls the built-in routine RDKEY at location $FD0C. This routine puts a flashing cursor on the screen and then jumps, indirectly, to the routine whose starting address is in locations $38 and $39, called KSWL (keyboard input switch low) and KSWH (keyboard input switch high). This is normally the address ($FD1B) of the routine KEYIN that waits for a key to be pressed and then places the ASCII code of the key (with bit 7 set) in accumulator A.

When the BASIC statement IN#*n* (or the monitor command *n* CTRL K) is executed (where *n* is the slot number), the address $C*n*00 is stored in locations $38 and $39 and the subroutine at $C*n*00 is executed. This routine can initialize the peripheral board, store a new entry address in locations $38 and $39, and then return to BASIC (or the monitor) by executing an RTS instruction. Subsequent INPUT statements will cause the routine on the peripheral board to be executed. This routine can input a character from a peripheral device. It should return with the ASCII code of the character (with bit 7 set) in accumulator A.

Making Peripheral Boards Slot Independent

If you want to make a peripheral board that will work in any slot (1–7), the on-board PROM program must do two things. First, it must be able to

determine which slot it is in. Second, no program instructions can refer to absolute addresses that are unique to a particular slot.

The program segment shown in Figure 13.10 can be used at the beginning of an on-board PROM to find the slot number. The subroutine

```
C200   20 4A FF   JSR SAVE        ;save registers
C203   78         SEI
C204   20 58 FF   JSR $FF58       ;(RTS)
C207   BA         TSX
C208   BD 00 01   LDA $0100, X    ;Cn
C20B   8D F8 07   STA $7F8        ;C2
C20E   29 0F      AND #$0F        ;0n
C210   A8         TAY             ;Y = $02
C211   0A         ASL
C212   0A         ASL
C213   0A         ASL
C214   0A         ASL             ;n0
C215   AA         TAX             ;X = $20
                  .
                  .
                  .
                  .
       20 3F FF   JSR RESTORE     ;restore registers
       60         RTS
```

FIGURE 13.10. Program segment to find slot number.

SAVE at $FF4A will store the contents of registers A, X, Y, CC, and SP in locations $45–$49, respectively. The subroutine RESTORE at $FF3F will restore these register values.

The starting address of the program shown in Figure 13.10 is $C200. This will be the address if the peripheral board is in slot 2. If the board is in slot *n*, this starting address will be $C*n*00. When the instruction JSR $FF58 at location $C204 is executed, the address $C206 is pushed on the stack, as shown in Figure 13.11. The subroutine at $FF58 just contains the RTS instruction, which will return to the instruction at $C207. However, the stack pointer now points to a memory location containing the high-order address C2, which contains the slot number 2. This value, in general C*n*, can be read by transferring the stack pointer to X and using the indexed mode of addressing. The value C*n* is stored in location $7F8, which is a location reserved for this slot number value. If this value is ANDed with $0F (at location $C20E) and TAY is executed, the index register Y will contain the slot number in the form $0*n*. Instructions such as STA $5F8,Y can be used to access the slot-dependent scratchpad RAM locations given in Table 13.3. Instructions from $C211–$C215 in Figure 13.10 will cause the slot number to be stored in index register X in the form $*n*0. Instructions such as LDA $C080,X can then be used to access the slot-dependent I/O locations given in Table 13.2.

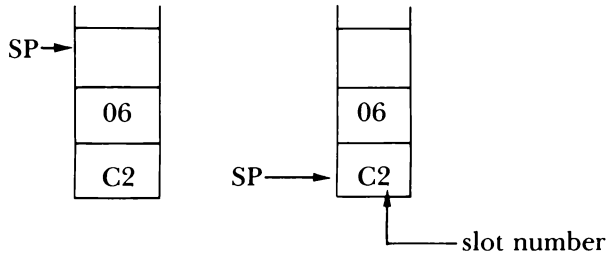


FIGURE 13.11. Address containing slot number is pushed on the stack.

Using Expansion ROMs

The $\overline{\text{I/O STROBE}}$ signal from output Z1 in Figure 13.4 goes low for any address in the range $C800–$CFFF. This signal is connected to pin 20 on *all* eight I/O slots. This signal can be used as a chip select signal for a 2K-byte PROM or ROM. This is in addition to the 256-byte PROM selected by the $\overline{\text{I/O SELECT}}$ signal.

Suppose that two or more peripheral boards have 2K-byte expansion ROMs. They all must share the same 2K address space between $C800 and $CFFF. How can you select only one of these ROMs at a time? Figure 13.12 shows one way to do this. When the 256-byte PROM in slot *n* is accessed, the $\overline{\text{I/O SELECT}}$ line for that slot will go low. This signal can be used to set a latch that will partially enable the 2K-byte ROM on that board. All 2K-byte ROMs on other peripheral boards will have similar

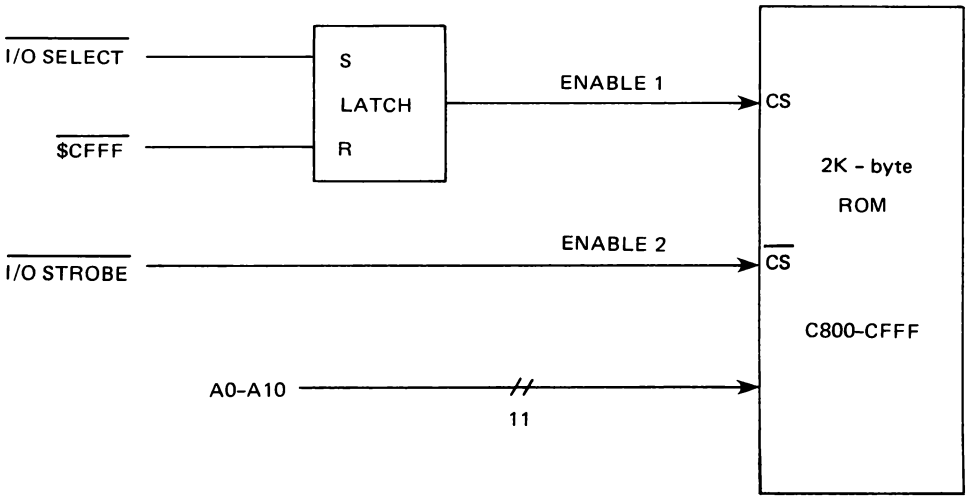


FIGURE 13.12. Method of accessing an expansion ROM.

latches. These latches are reset when the address $CFFF is put on the address bus. Therefore, if the instructions

C*n*XX	2C FF CF	BIT	$CFFF
C*n*XX + 03	4C 00 C8	JMP	$C800

are executed from the 256-byte PROM, the BIT $CFFF instruction will reset all latches and therefore turn off the 2K-byte ROMs on all peripheral boards (including the one you are trying to access). However, executing the next instruction at location $C*n*XX + 03 will cause the $\overline{\text{I/O SELECT}}$ line on that board to go low and set the latch in Figure 13.12. This will partially enable the ROM. It will be completely enabled when the address $C800 causes $\overline{\text{I/O STROBE}}$ to go low. The 2K-byte program starting at

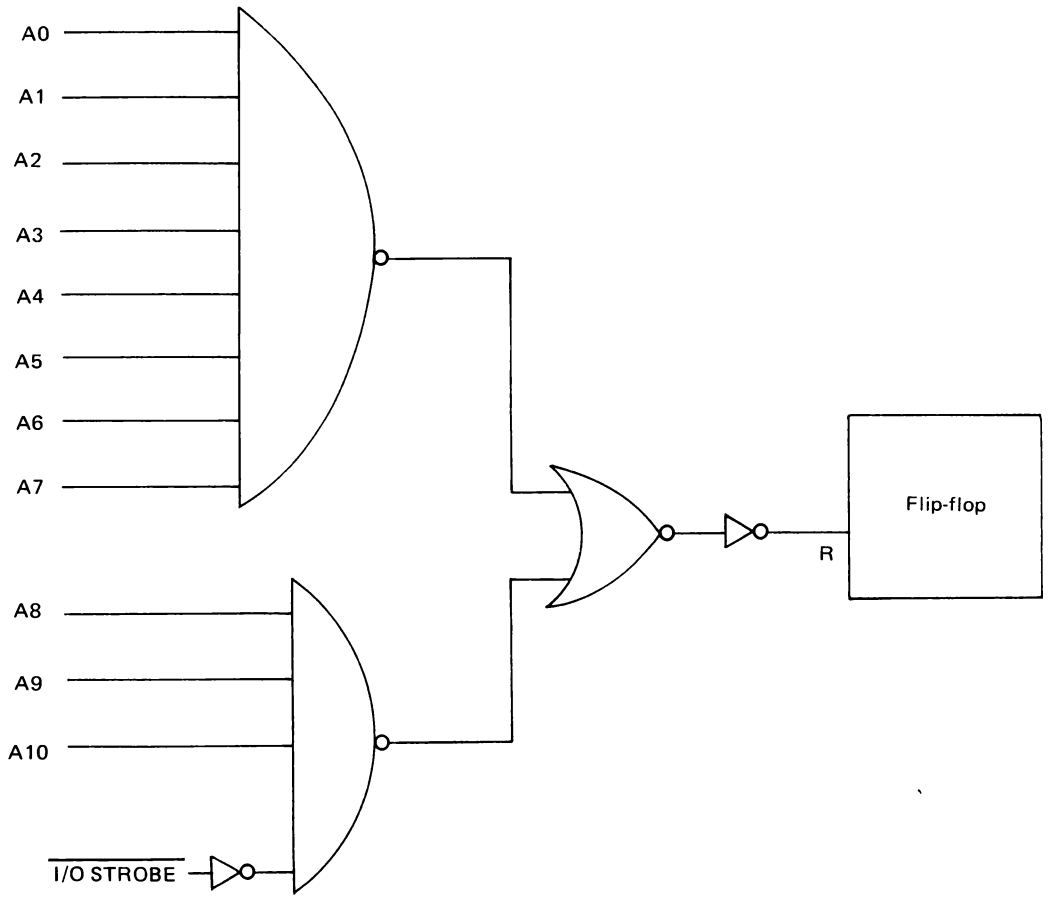


FIGURE 13.13. Decoding $CFFF.

location $C800 can now be executed without addressing expansion ROMs on other peripheral boards.

The only problem with the scheme is that you must decode the address $CFFF to apply to the RESET input of the latch. Additional chips will be required to decode this address completely. One way to do this is shown in Figure 13.13. Note that the output of a NAND gate is low only when all inputs are high, and the output of a NOR gate is high only when all inputs are low.

14

The 6821 Peripheral Interface Adapter (PIA)

The MC6821 (formerly the MC6820) is a peripheral interface adapter (PIA) manufactured by Motorola. The MCS6520 is a similar device manufactured by MOS Technology. The PIA is used to interface a 6502 (or Motorola 6800 family) microprocessor to peripheral devices via 8-bit parallel lines. The PIA can be connected to the Apple II through one of the I/O peripheral slots described in Chapter 13. In this chapter you will learn

1) how the PIA communicates with the microprocessor unit (MPU)
2) how to initialize a PIA
3) how interrupt signals are handled by the PIA
4) how the control lines of the PIA are programmed
5) how to put a PIA on an Apple II peripheral board
6) how to use a PIA to interface a printer to the Apple II

GENERAL DESCRIPTION OF A PIA

The PIA is a 40-pin chip whose pinout is shown in Figure 14.1. A general functional diagram of the PIA is shown in Figure 14.2. The lines at the bottom of this diagram connect the PIA to the MPU. The lines at the top of this diagram go to external peripheral devices. The data sheet for the PIA is given in Appendix D.

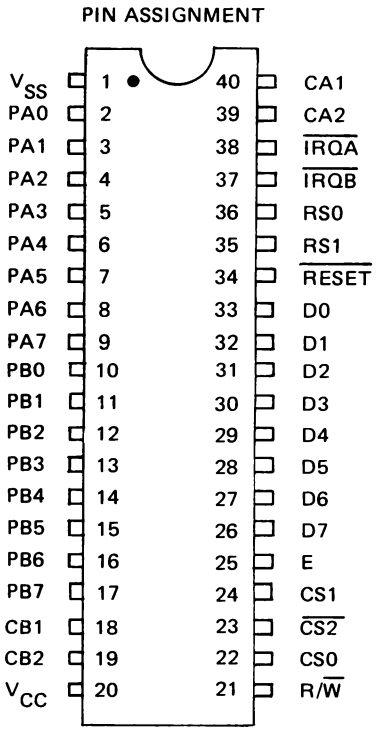


FIGURE 14.1. Pinout of a PIA.

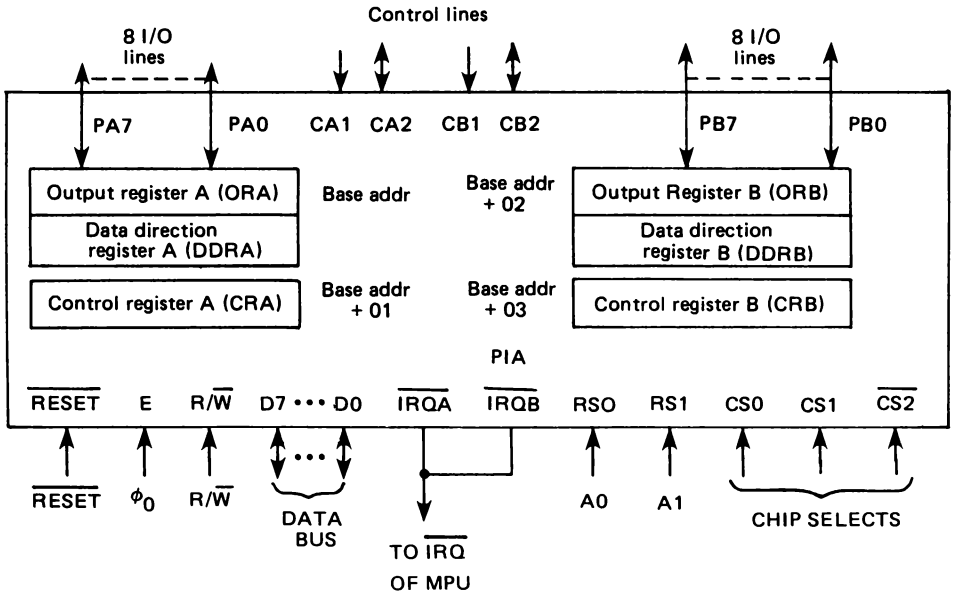


FIGURE 14.2. Functional diagram of a PIA.

The PIA is divided into two parts: an A side and a B side. These two sides are nearly identical, with only slight differences. Each side has an 8-bit output port and two control lines that can be connected to external devices. The lines of the 8-bit output ports can be individually programmed to be either input or output lines. These lines are labeled PA0–PA7 on the A side and PB0–PB7 on the B side. The two control lines on the A side are called CA1 and CA2, and the two control lines on the B side are called CB1 and CB2. CA1 and CB1 are always input lines, while CA2 and CB2 can be either input or output lines.

To the MPU the PIA looks like four memory locations. The PIA actually contains the six internal registers shown in Figure 14.2. However, the output register A (ORA) and the data direction register A (DDRA) share the same address. Similarly, the output register B (ORB) and the data direction register B (DDRB) share the same address. Which of these registers is addressed will depend on the state of bit 2 in the corresponding control register. For example, if *base addr* is the address of ORA and DDRA (see Figure 14.2), then this address will access ORA if bit 2 of CRA (at *base addr* + 01) is 1 and will access DDRA if bit 2 of CRA is 0. Similarly, *base addr* + 02 will access ORB if bit 2 of CRB is 1 and will access DDRB if bit 2 of CRB is 0. This is summarized in Table 14.1.

Each of the eight lines from output register A or output register B can be programmed to be either an input line or an output line. This is done by setting the corresponding bit in the data direction register. A 0-bit in the data direction register makes the corresponding line of the

Table 14.1 PIA Registers

	Address	*RS1*	*RS0*	*Bit 2 of CRA*	*Bit 2 of CRB*
Output register A (ORA)	*base addr*	0	0	1	
Data direction register A (DDRA)	*base addr*	0	0	0	
Control register A (CRA)	*base addr* + 01	0	1		
Output register B (ORB)	*base addr* + 02	1	0		1
Data direction register B (DDRB)	*base addr* + 02	1	0		0
Control register B (CRB)	*base addr* + 03	1	1		

output register an *input* line. A 1-bit in the data direction register makes the corresponding line of the output register an *output* line. For example, if DDRA contains 00001111, then PA0–PA3 are output lines and PA4–PA7 are input lines. Similarly, if DDRB contains 10101010, then PB0, PB2, PB4, and PB6 are input lines and PB1, PB3, PB5, and PB7 are out-

put lines. The data direction registers are set up at the beginning of the program and normally remain unchanged during the execution of the program.

The PIA is connected to the MPU through the data bus D0–D7. Address lines A0 and A1 from the MPU are connected to RS0 and RS1, the register select pins of the PIA. These select the four consecutive memory locations indicated in Table 14.1. The absolute value of *base addr* is determined by how the decoded address lines are connected to the chip select pins CS0, CS1, and $\overline{CS2}$. For example, if the $\overline{DEVICE\ SELECT}$ pin from slot 2 described in Chapter 13 is connected to $\overline{CS2}$, and if address lines A2 and A3 are connected to CS0 and CS1, respectively, the PIA registers will respond to addresses C0AC–C0AF.

THE CA1 AND CB1 CONTROL LINES

The control lines CA1 and CA2 are always input lines. They are used to detect high-to-low or low-to-high *transitions*. Which type of transition is active is determined by the setting of bit 1 in the control register. If bit 1 of control register A is 0, then a *high-to-low* transition on CA1 will cause bit 7 of control register A to be set to 1. If bit 1 of CRA is 1, then a *low-to-high* transition on CA1 will cause bit 7 of CRA to be set to 1. A similar situation occurs for CB1 and control register B (CRB). This is summarized in Figure 14.3.

Bits 6 and 7 of the control registers are *read only* bits. The only way that bit 7, the IRQA1 flag, can be set to 1 is by an active transition on CA1. Active means high-to-low or low-to-high depending upon the setting of bit 1. Since bit 7 is a read only bit it cannot be set to 0 by writing to the control register. The only way to clear bit 7 of CRA to 0 is to read output register A (ORA).* Similarly, the only way to clear bit 7 of CRB is to read ORB.

The two pins $\overline{IRQA}$ and $\overline{IRQB}$ shown at the bottom of Figure 14.2 are normally tied together and connected to the interrupt request pin $\overline{IRQ}$ of the MPU. When this line goes low, an interrupt is sent to the MPU. How the MPU processes interrupts will be described in Chapter 15. The interrupt pin $\overline{IRQA}$ on the PIA will go low on an active transition of CA1 if bit 0 of CRA is 1. If bit 0 of CRA is 0, the $\overline{IRQA}$ pin will remain high on an active transition of CA1. That is, the interrupts are masked and are not sent to the MPU. However, bit 7 of CRA always is set to 1 on an active transition of CA1. A similar situation applies to $\overline{IRQB}$ and CB1, as indicated in Figure 14.3.

* Bit 7 is also cleared when the $\overline{RESET}$ pin goes low.

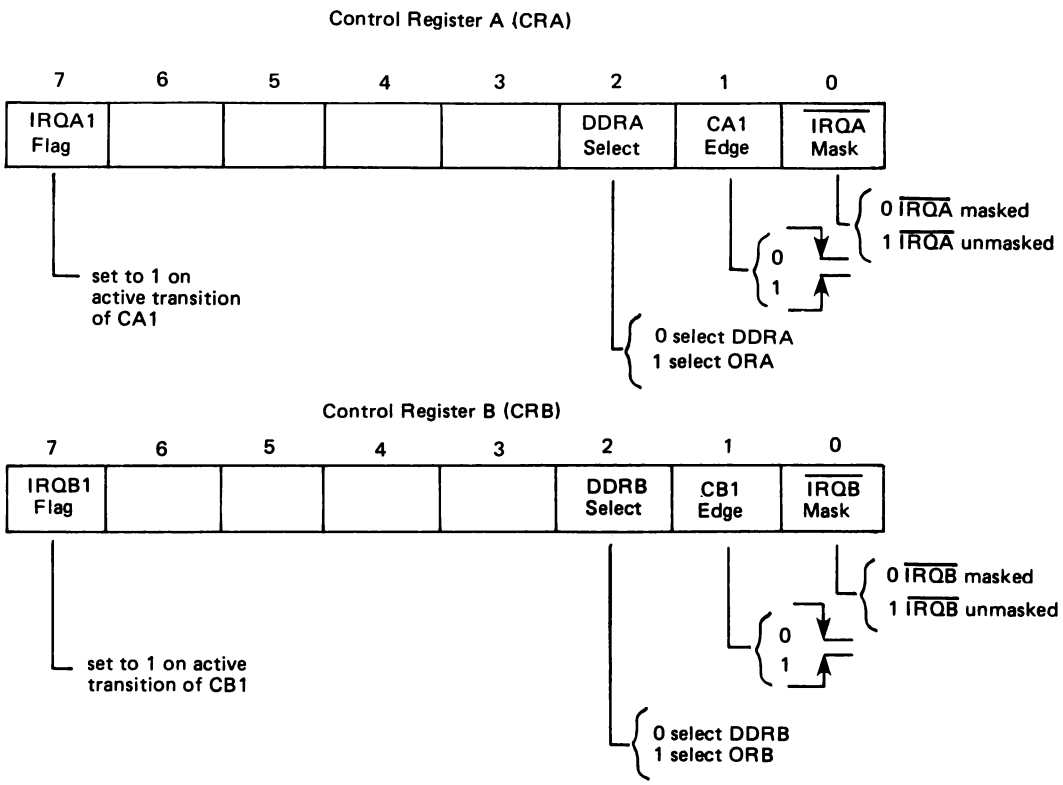


FIGURE 14.3. Bits of control registers A and B related to CA1 and CB1.

THE CA2 AND CB2 CONTROL LINES

The control lines CA2 and CB2 can be used as either input or output lines. If bit 5 of control register A (CRA) is 0, then CA2 is an *input* and bits 3 and 4 play the same role as bits 0 and 1 for CA1, as shown in Figure 14.4. Bit 6 of the control register is the interrupt flag that is set to 1 on an active transition of CA2 (or CB2). It plays the same role as bit 7 does for CA1 (or CB1). It is a read only bit that can only be reset to 0 by reading ORA (or ORB).

The control lines CA2 and CB2 are normally used as *output* lines. There are several different output modes, as shown in Figure 14.5. An output mode is indicated when bit 5 of the control register is 1. If bit 4 is also 1, the output mode is called the *follow mode.* This is because CA2 (or CB2) follows bit 3. That is, if bit 3 is low, CA2 (or CB2) is low. If bit 3 is high, CA2 (or CB2) is high. The follow mode is the most commonly used output mode.

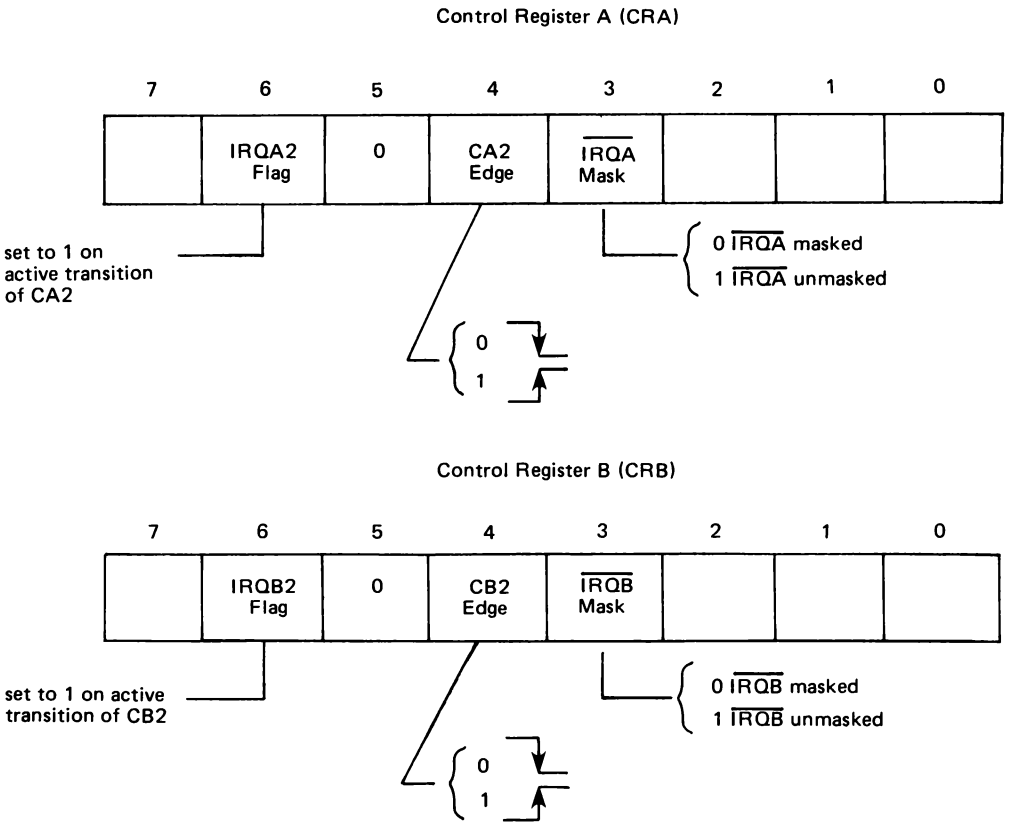


FIGURE 14.4. Bits of control registers A and B related to CA2 and CB2 used as inputs.

The Pulse Mode

In the pulse mode, CA2 (or CB2) is normally high. On the A side, CA2 will go low for one clock cycle after ORA has been read. For example, the instruction LDA ORA will cause a negative-going pulse one machine cycle wide to occur on CA2.

On the B side, CB2 will go low for one clock cycle after data have been written to ORB. For example, the instruction STA ORB will cause a negative-going pulse one machine cycle wide to occur on CB2.

The Handshake Mode

In the handshake mode, CA2 (or CB2) is normally low. It is expected that an interrupt signal will arrive on CA1 (or CB1). On the A side, an active transition of CA1 will cause CA2 to go high. It will remain high until ORA has been read. For example, the statement LDA ORA will cause CA2 to go low.

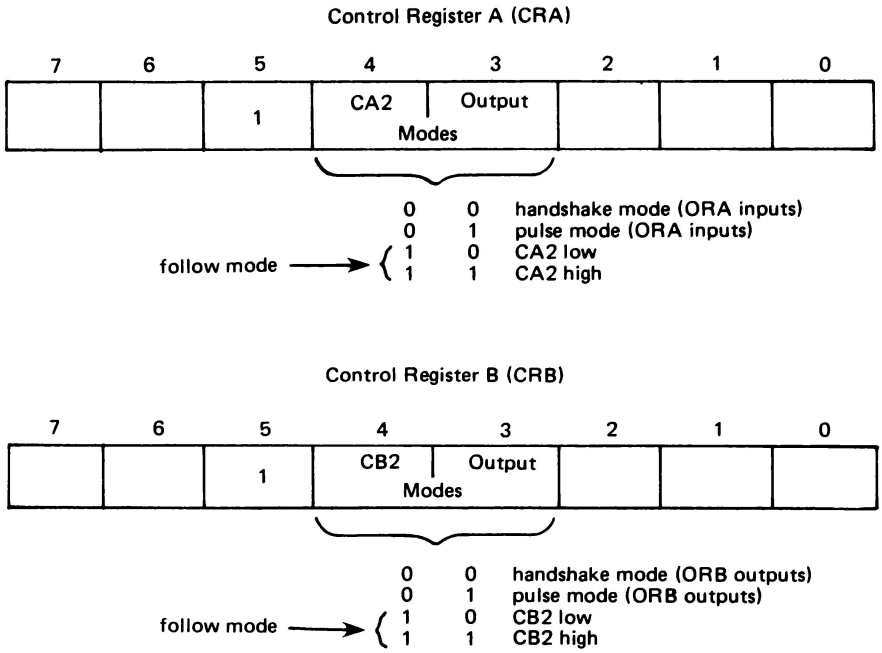


FIGURE 14.5. Bits of control registers A and B related to CA2 and CB2 used as outputs.

On the B side, an active transition of CB1 will cause CB2 to go high. It will remain high until data have been written to ORB. For example, the instruction STA ORB will cause CB2 to go low.

PUTTING A PIA ON AN APPLE II PERIPHERAL BOARD

The vector board no. 4609 shown in Figure 14.6 can be used to make your own Apple II peripheral boards. The vector board plugs directly into the slot connector shown in Figure 13.1. The PIA shown in Figure 14.1

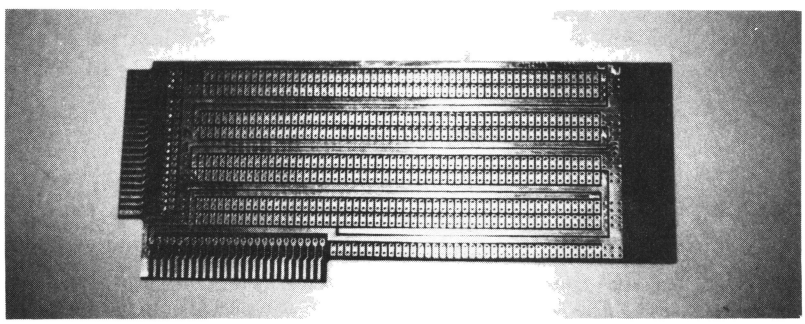


FIGURE 14.6. Vector board no. 4609 used for Apple II peripheral boards.

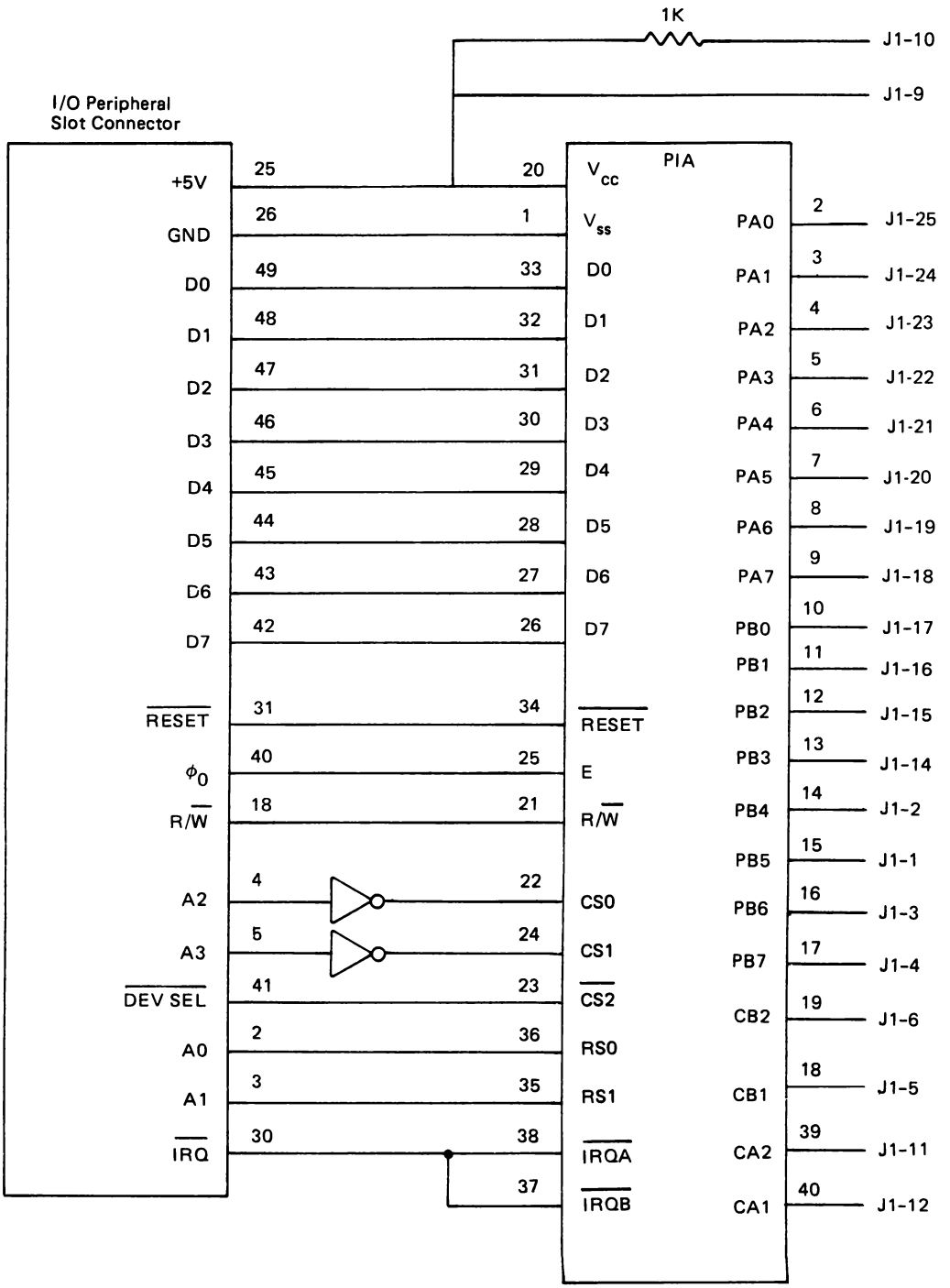


FIGURE 14.7. Schematic diagram of peripheral board PIA.

can be mounted on the board, using the schematic diagram shown in Figure 14.7.* A PIA mounted on such a board is shown in Figure 14.8. The output ports of the PIA are wired to a connector J1.

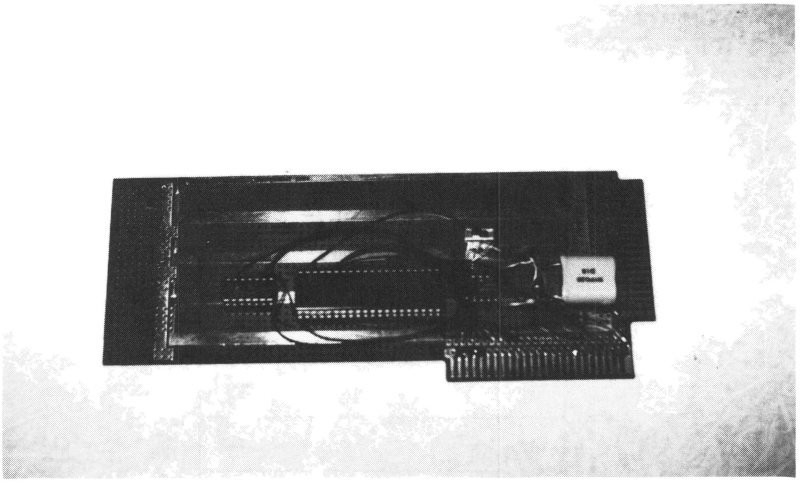


FIGURE 14.8. Peripheral board containing a PIA.

INTERFACING THE APPLE II WITH A PRINTER

The PIA will be used to interface the Apple II to the Anadex Model M-9500 printer shown in Figure 14.9. This printer has a parallel interface connector that inputs 8 data bits DB1–DB8. We will connect these to PB0–PB7 of the PIA. Only the 7 bits PB0–PB6 are used by the printer. These bits will contain the ASCII code of the character to be printed.

The printer has a $\overline{\text{DATA STROBE}}$ line that should be pulsed low after an ASCII code has been placed on the data lines DB1–DB7. We will connect this line to CB2 and use the follow mode to pulse this line low and then high again. This will cause the printer to print the character. When the printer has completed printing a character it will send back an

* You may sometimes have trouble connecting the $\overline{\text{DEV SEL}}$ output (pin 41) from the Apple I/0 slot to a chip select input of a peripheral device. The PIA data sheet (see Appendix D, Figures 11 and 12) states that the chip select inputs must be stable 160 nsec (t_{AS}) before the enable signal (E) goes high. We have E connected to ϕ_0 in Figure 14.7. But from Figure 13.5 in the last chapter, the $\overline{\text{DEVICE SELECT}}$ outputs won't go low until ϕ_1 goes low. The clock signal ϕ_1 is the complement of ϕ_0; therefore, ϕ_0 must go high (the enable signal) *before* the chip select signal from $\overline{\text{DEV SEL}}$ can go low. Although this violates the specifications in the data sheet, we have found that the PIA works anyway when wired as shown in Figure 14.7. Several suggestions have been made for overcoming this problem (see, for example, D. Paul and J. Wisman, COMPUTE!, August 1981, Issue 15, pp. 74–76).

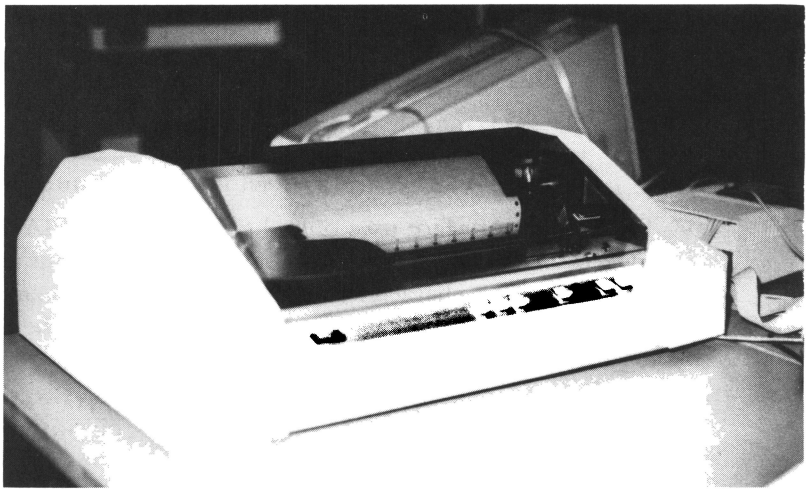


FIGURE 14.9. Anadex Model M-9500 printer interfaced through a PIA to an Apple II.

$\overline{\text{ACKNOWLEDGE}}$ signal to the PIA. This signal is normally high and goes low for approximately 4 μsec after a character has been printed. We will connect this signal to CB1 and program the PIA to sense a high-to-low transition.

The control register B (CRB) must therefore be initialized to $3C, as shown in Figure 14.10. Before this can be done, however, the value $FF must be stored in data direction register B (DDRB), so that ORB will be configured as an 8-bit *output* port. Therefore, the PIA is set up using the following instructions:

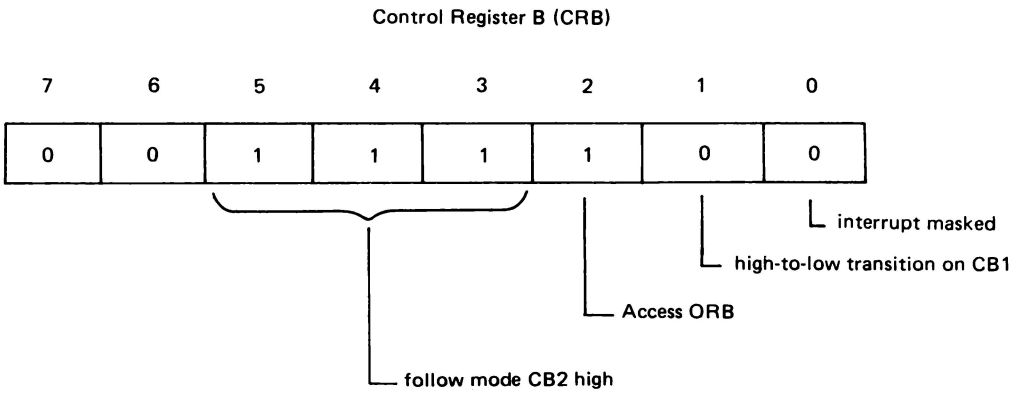


FIGURE 14.10. CRB is set to $3C for the printer interface.

```
        LDA   #$00      ;set bit 2 of CRB
        STA   CRB       ;to 0 to address DDRB
        LDA   #$FF
        STA   ORB       ;DDRB = $FF (ORB outputs)
        LDA   #$3C
        STA   CRB       ;set up CRB (Figure 14.10)
```

The printer interface uses only the B side of the PIA, so we have only initialized the B side. If you also use the A side, set up that side as well.

To send a character to the printer you must load accumulator A with the ASCII code of the character to be printed and then execute the following instructions:

```
        STA   ORB       ;put ASCII code on DATA lines DB1–DB7
        LDA   #$34      ;pulse DATA STROBE line low
        STA   CRB       ;by making bit 3 of CRB = 0
        LDA   #$3C      ;bring DATA STROBE high again
        STA   CRB       ;by setting bit 3 of CRB = 1
LOOP    BIT   CRB       ;wait for bit 7 (IRQB1 flag) of CRB to be
        BPL   LOOP      ;set to 1 as the result of ACK going low
        LDA   ORB       ;clear bit 7 of CRB (IRQB1 flag)
```

The first of these instructions stores the ASCII code of the character to be printed in ORB, which puts this code on the data lines DB1–DB7, connecting the PIA to the printer. The next four instructions pulse the DATA STROBE line low and then high. This tells the printer that it can start printing the character whose ASCII code is on the data lines DB1–DB7. The next two instructions form a loop that continues looping until the IRQB1 flag (bit 7) of CRB is set to 1. This will occur when the ACKNOWLEDGE signal from the printer, connected to CB1, goes low. The last statement clears this interrupt flag by reading ORB. The character has now been printed and the program can send another character to be printed.

Sending Characters from the Apple II to a Printer

In Chapter 13 we saw that Apple II statements that print characters on the screen generally call the built-in subroutine COUT ($FDED), which jumps, indirectly, to the subroutine whose address is stored in locations $36 (CSWL) and $37 (CSWH). When you boot a diskette by turning on the Apple II, the address $9EBD is stored in CSWL and CSWH. This is the address of the disk operating system (DOS) video intercept routine. This routine will check to see if the characters being sent to the screen represent a DOS command that must be executed. It will also execute COUT1 ($FDF0), which displays the character on the screen.

To send characters to the printer, the first step is to execute the "printer on" routine shown in Figure 14.11. This routine assumes that

```
                    0010  ;     PRINTER DRIVER
                          PROGRAM
                    0020  ;
                    0030  CRB     EQU C0A3
                    0040  ORB     EQU C0A2
                    0050  CSWL    EQU 36
                    0060  CSWH    EQU 37
                    0070  COUT1   EQU FDF0
                    0080  SLOTNO  EQU 302
                    0090  ;
                    0100          ORG 308
0308  A900      0110  PRTON   LDA #$00
030A  8DA3C0    0120          STA CRB
030D  A9FF      0130          LDA #$FF
030F  8DA2C0    0140          STA ORB
0312  A93C      0150          LDA #$3C
0314  8DA3C0    0160          STA CRB
0317  A940      0170          LDA #$40
0319  8536      0180          STA CSWL
031B  A903      0190          LDA #$03
031D  8537      0200          STA CSWH
031F  60        0210          RTS
```

FIGURE 14.11. Routine to turn the printer on.

the PIA is in slot 2. This routine first sets up the PIA as described in the previous section, and then stores the address of the main printer routine (given in Figure 14.13) in location $36 and $37. This printer routine will therefore be called every time the routine COUT is called.

To turn the printer off, execute the routine shown in Figure 14.12. This routine will replace locations $36 and $37 with the DOS video intercept address $9EBD.

```
0360   A9   9E   PRTOFF   LDA   #$9E
0362   85   37            STA   CSWH
0364   A9   BD            LDA   #$BD
0366   85   36            STA   CSWL
0368   60                 RTS
```

FIGURE 14.12. Routine to turn the printer off.

The main printer routine, starting at location $340, is shown in Figure 14.13. This routine first prints the character on the screen by calling COUT1 and then sends the character to the printer using the technique described in the previous section.

The TUTOR monitor has a built-in printer feature that allows you to print out disassembled versions of your programs. To do this, go to the

```
                    0220  ;
                    0230        ORG 340
                    0240  ;     MAIN PRINTER
                          ROUTINE
0340  20F0FD  0250  MAIN  JSR COUT1
0343  8DA2C0  0260        STA ORB
0346  A934    0270        LDA #$34
0348  8DA3C0  0280        STA CRB
034B  A93C    0290        LDA #$3C
034D  8DA3C0  0300        STA CRB
0350  ADA3C0  0310  LOOP  LDA CRB
0353  10FB    0320        BPL LOOP
0355  ADA2C0  0330        LDA ORB
0358  60      0340        RTS
```

FIGURE 14.13. Main printer routine.

first byte of the program you wish to disassemble. Then type /P. The message

PRINTER: A OR SLOT #

will appear on the command line. If you are using a printer interface board that contains an on-board PROM, type the slot number *n*. This will use the address C*n*00 as the address of the routine to turn the printer on.

If you are using your own printer routine, type A. The message

ENTER DRIVER ADDRESS

will appear on the command line. Type in the address of your printer set-up routine. This would be 308 using the routine in Figure 14.11. It must be the address of the setup routine that stores the address of the main driver program in $36 and $37.

After you have entered either the slot number or the address of your own routine, the message

LIST: P B S

will appear on the command line. This is the same LIST command described in Chapter 9 (see Figure 9.10). In this case the disassembled instructions will not only be printed on the screen but will also be sent to the printer. The TUTOR monitor automatically turns off the printer after the instructions have been printed.

15

Interrupts

Interrupts are used to suspend the normal execution of a program, usually in response to an external signal. When this occurs control is transferred to a special part of the program called an *interrupt service routine.* After you have executed this interrupt service routine, control returns to the point at which the program was interrupted. In this chapter you will learn

1. how different types of 6502 interrupts are generated
2. where to store interrupt vectors
3. how interrupts are processed by the 6502
4. how to use interrupts with a PIA
5. how to display a real time clock on the Apple II

6502 INTERRUPTS

The 6502 microprocessor can handle four different types of interrupts—reset, maskable interrupt, nonmaskable interrupt, and software interrupt or break. The first three are hardware interrupts that occur when a particular pin on the 6502 chip goes low. The last one is a software interrupt that occurs when the BREAK instruction (op-code = 00) is executed.

Reset

Pin 40 of the 6502 is the $\overline{\text{RESET}}$ pin (see Figure 2.3). When this pin goes low, normal microprocessor functions are suspended. When a positive edge on this pin is detected, the microprocessor will set the interrupt flag in the status (condition code) register and then load the program counter with the contents of locations $FFFC and $FFFD.

The Apple II has a power-up reset circuit that causes this reset pin to remain low until after power (5 volts) has been applied to pin 8 of the 6502. Locations $FFFC and $FFFD in the Apple II Autostart ROM contain the reset vector address $FA62. Execution starts at this address every time you press the reset key (or turn the power on).

This reset program can tell the difference between a power-up reset (cold start) and pressing the reset key (warm start). It does this by comparing the contents of $3F3 (exclusive-ORed with $A5) with the contents of location $3F4. If no match occurs it must be a power-up condition, because the values in these locations will be random. In this case the screen is cleared. The address of the program to branch to on subsequent resets is stored in locations $3F2 and $3F3 and the exclusive-OR of the contents of $3F3 and $A5 is stored in location $3F4. The next time you press the RESET key, a match with location $3F4 will occur, indicating a warm start, and the screen will not be cleared.

You can change the program that is executed when you press the reset key by changing the contents of locations $3F2–$3F4. We have stored the address $8000 in locations $3F2 and $3F3 and have stored the value $25 ($80 EOR $A5) in location $3F4. This is why the TUTOR monitor (starting at location $8000) is executed every time you press the reset key.

Maskable Interrupts

Pin 4 of the 6502 is the $\overline{\text{IRQ}}$ pin. It is connected to pin 30 of each of the eight peripheral I/O slots. When this line goes low it initiates an interrupt sequence. When the current instruction is completed, the interrupt flag I in the status (or condition code) register is checked. If this bit is set to 1, interrupts are masked and the normal execution of the program will be continued. On the other hand, if the interrupt flag I is cleared to 0, the interrupt sequence will proceed.

The program counter and condition code register will be saved on the stack, as shown in Figure 15.1. The interrupt flag I in the condition code register will then be set to 1 to prevent another interrupt from occurring. Next, the program counter will be loaded with the vector address stored in locations $FFFE and $FFFF (see Figure 15.2). This means that the starting address of the interrupt service routine must be stored in locations $FFFE and $FFFF.

In the Apple II Autostart ROM the starting address $FA40 is stored in locations $FFFE and $FFFF. This interrupt service routine will jump to

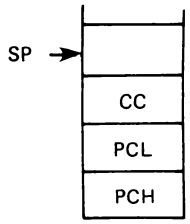


FIGURE 15.1.
The program counter and condition code register are saved on the stack when an interrupt is processed.

a user interrupt service routine whose starting address is stored in locations $3FE (low byte) and $3FF (high byte). Thus, if you write an interrupt service routine to respond to an $\overline{\text{IRQ}}$ signal, you must store the starting address of this routine in locations $3FE and $3FF.

The last statement in an interrupt service routine must be the RTI (return from interrupt) instruction (op-code = 40). This statement pulls the condition code register and the program counter off the stack. The program will therefore continue at the point in the program where the interrupt occurred.

Only the program counter and the condition code register are saved on the stack when an interrupt occurs. If your interrupt service routine uses A, X, and Y, you must also save these values by pushing them on the stack. You can do this with the following instructions:

```
PHA     ;Push A
TXA     ;X → A
PHA     ;Push it
TYA     ;Y → A
PHA     ;Push it
```

At the end of the interrupt service routine you must pull them off the stack using the following instructions:

```
PLA     ;Pull Y
TAY     ;Restore it
PLA     ;Pull X
TAX     ;Restore it
PLA     ;Restore A
RTI     ;Return from interrupt
```

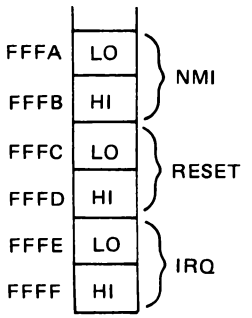


FIGURE 15.2.
Vector addresses associated with 6502 interrupts.

Nonmaskable Interrupts

Pin 6 of the 6502 microprocessor is the nonmaskable interrupt ($\overline{NMI}$) pin. It is connected to pin 29 of each of the peripheral I/O connectors. When this pin goes low an interrupt sequence similar to the $\overline{IRQ}$ sequence is initiated. The differences are that the I flag in the status register *cannot* mask this interrupt, and the starting address of the interrupt service routine must be stored in locations $FFFA and $FFFB, as shown in Figure 15.2. The Apple II autostart ROM stores the address $03FB in locations $FFFA and $FFFB. You can store a JMP instruction to your nonmaskable interrupt service routine in locations $3FB-$3FD.

The BREAK Instruction

The BRK instruction (op-code = 00) is a "software" interrupt. When this instruction occurs, the program saves the status register and program counter on the stack and branches to the address stored in locations $FFFE and $FFFF. This is the same sequence of events that occurs when the $\overline{IRQ}$ hardware interrupt occurs. The difference is that the B-flag (bit 4) in the status register is set to 1 when the BRK instruction is executed (see Figure 15.3). This B-flag must be checked in the interrupt service

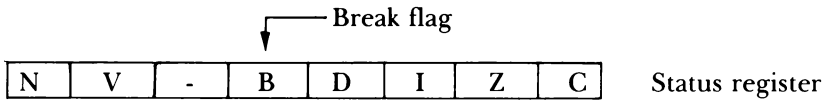


FIGURE 15.3. The BRK instruction sets the B-flag in the status register.

routine to determine if the interrupt is a BRK instruction or a hardware ($\overline{IRQ}$) interrupt. This is done in the Apple II (autostart ROM) by using the program segment shown in Figure 15.4. The starting address of IRQ is $FA40; this is the address that is stored in $FFFE and $FFFF.

Note that when an interrupt occurs, accumulator A is stored in location $45* and bit 4 of the status register is checked. If the B-flag is 1, the program branches to BREAK (software interrupt); otherwise it jumps to the address stored in locations $3FE and $3FF (hardware interrupt). If the interrupt is the result of a BRK instruction, the registers X, Y, CC, and SP are saved in locations $46–$49 and the program jumps to the address stored in locations $3F0 and $3F1.

* You must restore the value of accumulator A by executing LDA $45 just before the RTI instruction in your interrupt service routine.

```
FA40   85 45      IRQ     STA   ACC
FA42   68                 PLA
FA43   48                 PHA
FA44   0A                 ASL   A
FA45   0A                 ASL   A
FA46   0A                 ASL   A
FA47   30 03              BMI   BREAK
FA49   63 FE 03           JMP   (IRQLOC)
FA4C   28         BREAK   PLP
FA4D   20 4C FF           JSR   SAV1
FA50   68                 PLA
FA51   85 3A              STA   PCL
FA53   68                 PLA
FA54   85 3B              STA   PCH
FA56   6C F0 03           JMP   (BRKV)

FF4C   86 46      SAV1    STX   XREG
FF4E   84 47              STY   YREG
FF50   08                 PHP
FF51   68                 PLA
FF52   85 48              STA   STATUS
FF54   BA                 TSX
FF55   86 49              STX   SPNT
FF57   D8                 CLD
FF58   60                 RTS
```

FIGURE 15.4. Interrupt service routine executed by Apple II Plus with autostart ROM.

The BRK instruction is used by the TUTOR monitor to set breakpoints and execute single-step instructions.

USING INTERRUPTS WITH A PIA

Recall from Chapter 14 that an active transition on CA1 (or CB1) will set bit 7 of CRA (or CRB) to 1. At the same time, if bit 0 of CRA (or CRB) is set to 1, the $\overline{\text{IRQA}}$ (or $\overline{\text{IRQB}}$) pin of the PIA will go low. If this pin is connected to the $\overline{\text{IRQ}}$ pin of the 6502, an interrupt will occur (assuming that the I bit of the status register has been cleared by the CLI instruction) on an active transition of the PIA's CA1 (or CB1) control line.

When setting up an Apple II program that uses interrupts, you must store the starting address of the interrupt service routine in locations $3FE and $3FF. You must also set bit 0 of the PIA control register to 1, which will unmask the interrupt to the MPU. Finally, you must remember to clear the interrupt flag of the status register by executing the CLI instruction. The general form of a program that uses interrupts is shown in Figure 15.5.

```
MAIN    SEI                 ;set interrupt mask
        -
        -
        JSR     PIASU       ;set up PIA with bit 0 of control register set to 1
        -
        -
        -
        LDA     #$54  }     store interrupt service routine address $0354 in
        STA     $3FE  }     locations $3FE and $3FF
        LDA     #$03  }
        STA     $3FF  }
        -
        -
        CLI                 ;clear interrupt mask
LOOP    -             }
        -             }
        -             }     interrupts respond here
        -             }
        JMP     LOOP  }
* INTERRUPT SERVICE ROUTINE
  0354          -
        -
        LDA     $45
        RTI
```

FIGURE 15.5. Using interrupts with a PIA on an Apple II.

Real-Time Clock

The circuit shown in Figure 15.6 will produce a 60-Hz signal that can be connected to the CA1 control line of a PIA. The PIA will be set up to produce an interrupt every 1/60 second. By having the interrupt service rou-

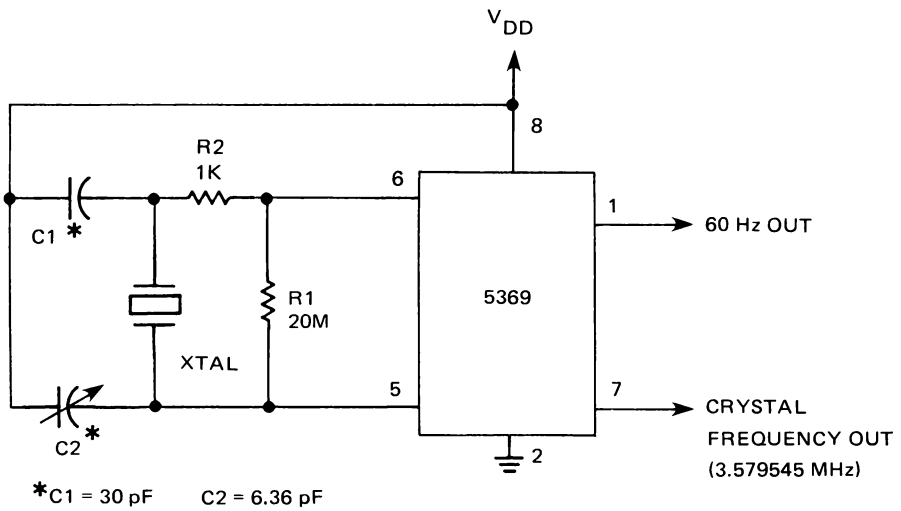


FIGURE 15.6. Circuit for producing a 60-Hz signal.

tine count these interrupts, a real-time digital clock can be displayed on the video screen. The program shown in Figure 15.7 will let the user enter the current time (hours mins secs) and will then display the time at the upper left-hand corner of the screen in the format 11:25:38, while the TUTOR monitor is being used in the normal way.

FIGURE 15.7. Program to display a real-time digital clock on the Apple II video screen.

```
                0010  ;       REAL TIME CLOCK
                0020  MONIT   EQU 8000
                0030  KEYIN   EQU 8056
                0040  COUT1   EQU FDF0
                0050  BASCLC  EQU FBC1
                0060  CH      EQU 24
                0070  ACC     EQU 45
                0080          ORG 93F0
                0090  HOURS   EQU *
                0100  MINS    EQU *+1
                0110  SECS    EQU *+2
                0120  COUNT   EQU *+3
                0130  BYTE    EQU *+4
                0140  ORA     EQU C0A0
                0150  CRA     EQU C0A1
                0160  INTVEC  EQU 3FE
                0170  VIDSRT  EQU 400
                0180          ORG 9400
                0190  ;
                0200  ;       MAIN PROGRAM
9400  A935      0210  CLOCK   LDA #$35
9402  8DA1C0    0220          STA CRA         ;enable interrupts
9405  AD4E94    0230          LDA INTSV
9408  8DFE03    0240          STA INTVEC      ;store interrupt vector
940B  AD4F94    0250          LDA INTSV+01
940E  8DFF03    0260          STA INTVEC+01
9411  A93C      0270          LDA #$3C
9413  8DF393    0280          STA COUNT       ;COUNT=60
9416  A917      0290          LDA #$17
9418  20C1FB    0300          JSR BASCLC
941B  A900      0310          LDA #$00
941D  8524      0320          STA CH
941F  203594    0330          JSR GETBYT      ;enter current
9422  8DF093    0340          STA HOURS       ;  hours
9425  203594    0350          JSR GETBYT
9428  8DF193    0360          STA MINS        ;  minutes
942B  203594    0370          JSR GETBYT
942E  8DF293    0380          STA SECS        ;  seconds
9431  58        0390          CLI             ;clear interrupt mask
9432  4C0080    0400          JMP MONIT       ;go to TUTOR
                0410  ;
```

```
                0420  ;       GET BYTE
9435  205680    0430  GETBYT  JSR KEYIN       ;get high nibble
9438  20F0FD    0440          JSR COUT1       ;display it
943B  0A        0450          ASL
943C  0A        0460          ASL
943D  0A        0470          ASL
943E  0A        0480          ASL
943F  8DF493    0490          STA BYTE
9442  205680    0500          JSR KEYIN       ;get low nibble
9445  20F0FD    0510          JSR COUT1       ;display it
9448  290F      0520          AND #$0F
944A  0DF493    0530          ORA BYTE
944D  60        0540          RTS
                0550  ;
944E  5094      0560  INTSV   EQD INTSER
                0570  ;       INTERRUPT SERVICE ROUTINE
9450  98        0580  INTSER  TYA
9451  48        0590          PHA             ;save Y
9452  8A        0600          TXA
9453  48        0610          PHA             ;save X
9454  18        0620          CLC
9455  F8        0630          SED             ;do decimal arithmetic
9456  CEF393    0640          DEC COUNT       ;if not 60 counts
9459  D059      0650          BNE OUT2        ;then exit
945B  ADF293    0660          LDA SECS        ;else
945E  6901      0670          ADC #$01        ;add 1 to seconds
9460  8DF293    0680          STA SECS
9463  C960      0690          CMP #$60        ;after 59 seconds
9465  D027      0700          BNE OUT1
9467  A900      0710          LDA #$00        ;reset seconds to 0
9469  8DF293    0720          STA SECS
946C  ADF193    0730          LDA MINS        ;add 1 to minutes
946F  6900      0740          ADC #$00        ;(carry set)
9471  8DF193    0750          STA MINS
9474  C960      0760          CMP #$60        ;after 59 minutes
9476  D016      0770          BNE OUT1
9478  A900      0780          LDA #$00        ;reset minutes to 0
947A  8DF193    0790          STA MINS
947D  ADF093    0800          LDA HOURS       ;add 1 to hours
9480  6900      0810          ADC #$00        ;(carry set)
9482  8DF093    0820          STA HOURS
9485  C913      0830          CMP #$13        ;after 12 hours
9487  D005      0840          BNE OUT1
9489  A901      0850          LDA #$01        ;set hours to 1
948B  8DF093    0860          STA HOURS
948E  D8        0870  OUT1    CLD             ;do binary arithmetic
948F  A000      0880          LDY #$00
9491  ADF093    0890          LDA HOURS
9494  20BE94    0900          JSR PRBYTE      ;display hours
9497  A9BA      0910          LDA #":
```

```
9499  990004  0920          STA VIDSRT,Y   ;display :
949C  C8      0930          INY
949D  ADF193  0940          LDA MINS
94A0  20BE94  0950          JSR PRBYTE     ;display minutes
94A3  A9BA    0960          LDA #":
94A5  990004  0970          STA VIDSRT,Y   ;display :
94A8  C8      0980          INY
94A9  ADF293  0990          LDA SECS
94AC  20BE94  1000          JSR PRBYTE     ;display seconds
94AF  A93C    1010          LDA #$3C
94B1  8DF393  1020          STA COUNT      ;set COUNT=60
94B4  ADA0C0  1030  OUT2    LDA ORA        ;clear interrupt flag
94B7  68      1040          PLA
94B8  AA      1050          TAX            ;restore X
94B9  68      1060          PLA
94BA  A8      1070          TAY            ;restore Y
94BB  A545    1080          LDA ACC        ;restore A
94BD  40      1090          RTI
              1100  ;
              1110  ;       PRINT BYTE
94BE  48      1120  PRBYTE  PHA            ;save A
94BF  4A      1130          LSR
94C0  4A      1140          LSR
94C1  4A      1150          LSR
94C2  4A      1160          LSR
94C3  09B0    1170          ORA #$B0
94C5  990004  1180          STA VIDSRT,Y   ;display upper nibble
94C8  68      1190          PLA            ;restore A
94C9  290F    1200          AND #$0F       ;mask upper nibble
94CB  09B0    1210          ORA #$B0
94CD  C8      1220          INY
94CE  990004  1230          STA VIDSRT,Y   ;display lower nibble
94D1  C8      1240          INY
94D2  60      1250          RTS
```

16

Serial I/O—The ACIA

In Chapter 14 you learned how to use a PIA to interface the 6502 microprocessor to external devices through an 8-bit parallel port. The MC6850, called an *asynchronous communications interface adapter* (ACIA), is used to interface a microprocessor to external devices by means of a serial I/O line. In this case data are transferred 1 bit at a time rather than 1 byte (8 bits) at a time. In this chapter you will learn

1. how asynchronous serial data are transmitted and received
2. how the ACIA can be used to send and receive serial data
3. how to use an ACIA with an Apple II
4. how to use the Apple II as a computer terminal

ASYNCHRONOUS SERIAL DATA

Parallel I/O requires eight data lines to transmit a byte (8 bits) of data. This becomes expensive when data have to be sent a long distance. It is much less expensive to use a single data line over which the 8 bits are sent 1 bit at a time. This is called *serial communication*; it will obviously be considerably slower than sending the 8 bits in parallel.

Asynchronous serial communication uses a start bit to tell when a

particular character is being sent. This is illustrated in Figure 16.1, which shows the transmitted waveform when the character T (ASCII code = $54)

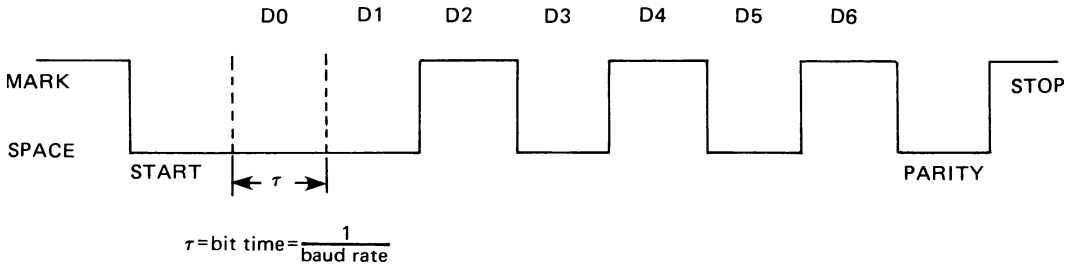


FIGURE 16.1. ASCII code $54 = 1010100 (T) sent with odd parity.

is sent with odd parity. Before a character is sent the line is in the high or *mark* state. The line is then brought low (called a *space*) and held low for a time τ, called the *bit time*. This first space is called the *start bit*. It is typically followed by 7 or 8 data bits. The least significant bit D0 is transmitted first. For example, in Figure 16.1, the 7 bits corresponding to the ASCII code $54 (the character T) are sent, starting with D0. These 7 bits are followed by a *parity bit*. This bit is set to a 1 or a 0 so that the sum of the number of 1s transmitted is either even or odd. We have used odd parity in Figure 16.1. Since three 1s were sent (D2, D4, and D6), the parity bit is 0. The parity bit is followed by 1 or 2 stop bits, which are always high (a mark). The next character will be indicated by the presence of the next start bit.

The reciprocal of the bit time is called the *baud rate*. Common baud rates used in serial communication are given in Table 16.1. The 110-baud rate uses 2 stop bits. This means that 11 bit times per character are used (7 data bits + 1 parity bit + 1 start bit + 2 stop bits). Therefore, 10 characters per second (cps) are transmitted at 110 baud. The remaining baud rates in Table 16.1 use 1 stop bit and therefore use 10 bit times per character.

Table 16.1 Common Asynchronous Serial Baud Rates

Baud Rate	*Bit Time (msec)*	*No. of STOP Bits*	*Char. Time (msec)*	*cps*
110	9.09	2	100	10
300	3.33	1	33.3	30
600	1.67	1	16.7	60
1,200	0.833	1	8.33	120
2,400	0.417	1	4.17	240
9,600	0.104	1	1.04	960

The ACIA is a 24-pin chip whose pinout is shown in Figure 16.2. The data sheet for the ACIA is given in Appendix D. A general functional dia-

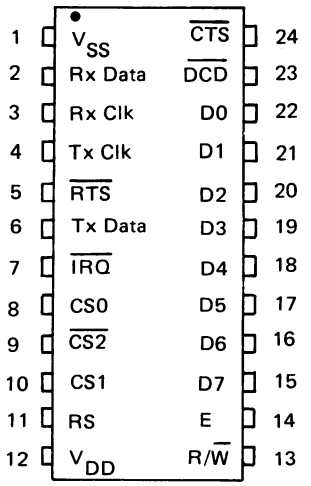


FIGURE 16.2. Pinout of an ACIA.

gram of the ACIA is shown in Figure 16.3. The lines at the bottom of this diagram connect the ACIA to the MPU. The lines at the right go to external devices. The two lines at the left are transmit and receive clock inputs, which determine the baud rate.

The main function of the ACIA is to transform parallel data from the MPU into serial data and send them out through the transmit data pin TxD, and to receive serial data through the receive pin RxD and transform them to parallel data that can be read by the MPU.

To the MPU the ACIA looks like two memory locations. The ACIA actually contains four internal registers that can be accessed by the MPU. The control register (CR) and the status register (SR) share the same address (*base addr*). They are distinguished by the state of the read/write line. The control register is a write only register and the status register is a read only register. Thus, writing to the *base addr* will store data in the control register, while reading from the *base addr* will load data from the status register.

The transmit data register (TDR) and the receive data register (RDR) also share the same address (*base addr* + 01). The transmit data register is a write only register and the receive data register is a read only register. Address line A0 is connected to the register select pin RS; together with the read/write line, it determines which register is selected. This is summarized in Table 16.2.

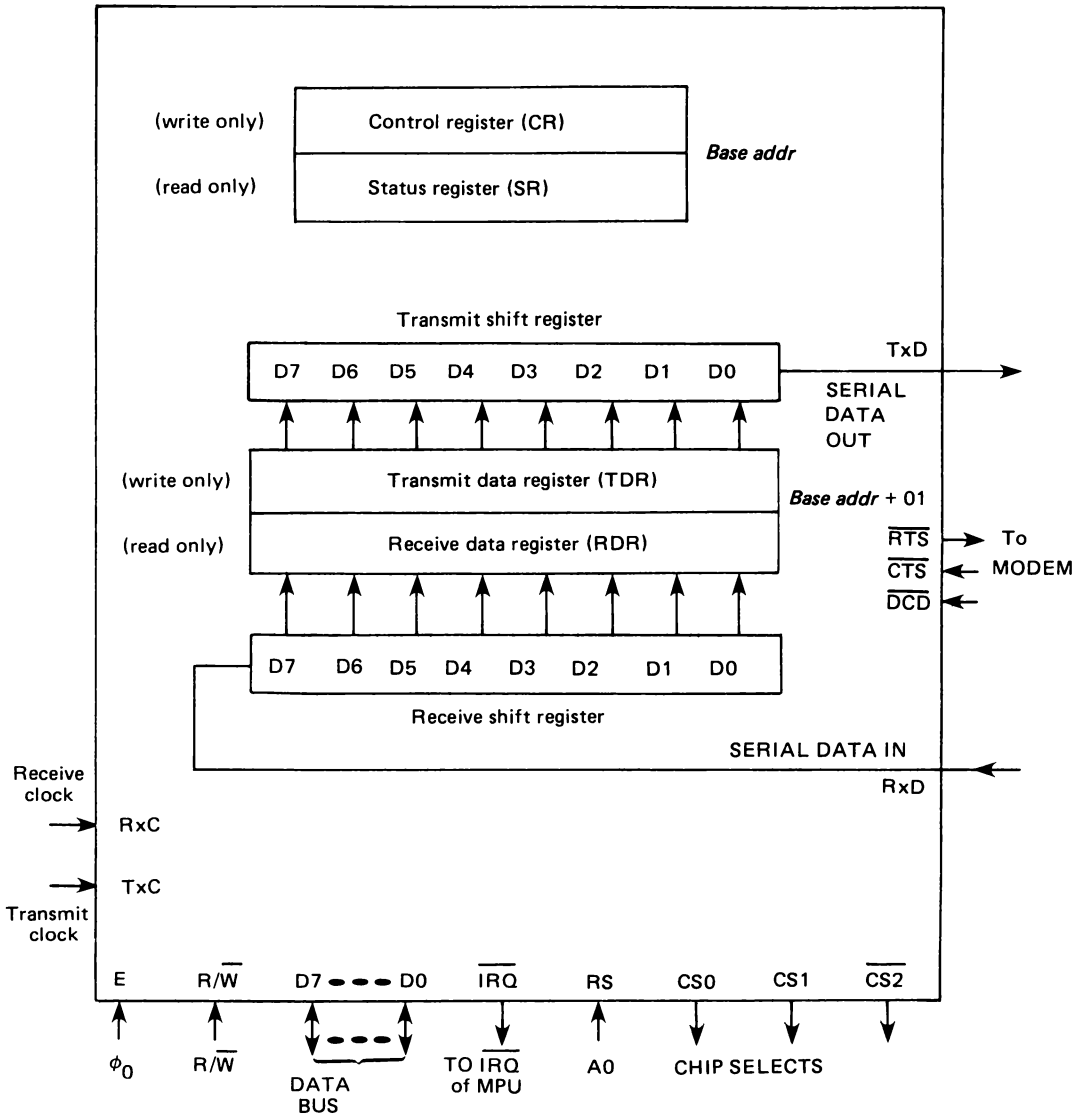


FIGURE 16.3. Functional diagram of an ACIA.

Table 16.2 ACIA Registers

	Address	*RS*	$R/\overline{W}$
Control register (CR)	*base addr*	0	0
Status register (SR)	*base addr*	0	1
Transmit data register (TDR)	*base addr* + 01	1	0
Receive data register (RDR)	*base addr* + 01	1	1

The Control Register

When using an ACIA you must first set up the control register shown in Figure 16.4. The two bits CR0 and CR1 perform two different functions, serving as a master reset and as a counter divide select, according to Fig-

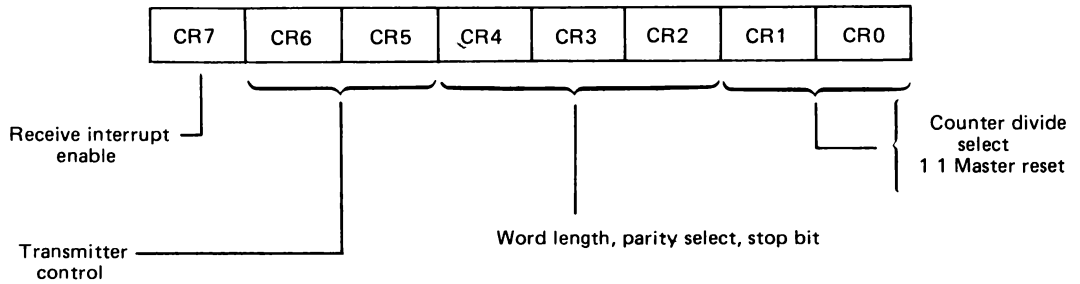


FIGURE 16.4. The ACIA control register.

ure 16.5. Before using the ACIA you must reset it by storing the bit pattern 11 in bits 0 and 1 of the control register.

CR1	*CR0*	*Function*
0	0	÷ 1
0	1	÷ 16
1	0	÷ 64
1	1	Master reset

FIGURE 16.5. Counter divide select bits CR0, CR1.

You then store the proper counter divide code in these bits. This value (1, 16, or 64) is the value by which the receive and transmit clock inputs are divided to obtain the baud rate. For example, if you want to receive data at 300 baud and you set CR1 = 0 and CR0 = 1 (÷ 16), then the receive clock input RxC must be 300 × 16 = 4,800 cps. You must use a clock frequency higher than the baud rate so that the ACIA can locate the center of the bit times. For example, suppose that the counter divide is set to ÷ 16. When a start bit is detected on the receive line (a high-to-low transition), the receive clock counts to 8. This should be near the center of the start bit. (Remember that the clock rate is 16 times the baud rate.) If the line is still low, it must be a valid start bit (rather than a noise glitch). The receive clock then counts to 16, which should locate the center of the first data bit D0. This value can then be read by the ACIA. Sixteen clock pulses later the value of the second data bit D1 can be read. This process continues until the entire byte is read. Note that if the baud rate is slightly off, the ACIA may still be able to read the correct value as long as the values read every 16 clock pulses do not drift outside the data bit cells. Using a ÷ 64 clock would result in reading even closer to the center of the individual data bit times.

Bits 2, 3, and 4 of the control register in Figure 16.4 are used to select the word length, parity, and number of stop bits, according to Figure 16.6. The number of bits in this figure is the number of *data* bits, not counting the parity bit. For example, to send a standard 7-bit ASCII code with odd parity and 1 stop bit, the bit pattern 011 would be stored in CR4, CR3, and CR2.

CR4	*CR3*	*CR2*	*No. of bits*	*Parity*	*No. of stop bits*
0	0	0	7	EVEN	2
0	0	1	7	ODD	2
0	1	0	7	EVEN	1
0	1	1	7	ODD	1
1	0	0	8	NONE	2
1	0	1	8	NONE	1
1	1	0	8	EVEN	1
1	1	1	8	ODD	1

FIGURE 16.6. Word length, parity, and stop bit select bits CR2, CR3, and CR4.

Bits 5 and 6 of the ACIA control register are transmitter control bits. The four possible combinations of these bits are given in Figure 16.7. These bits are used to enable or disable transmitting interrupts, to set the level of the $\overline{\text{RTS}}$ ($\overline{\text{request-to-send}}$) output, and to send a BREAK level (space) on the transmit line.

Bit 7 of the ACIA control register is used to enable or disable receive interrupts according to Figure 16.8. Interrupts can occur when various bits in the ACIA status register are set.

CR6	*CR5*	$\overline{RTS}$	*Transmitting Interrupt*	
0	0	LOW	DISABLED	
0	1	LOW	ENABLED	
1	0	HIGH	DISABLED	
1	1	LOW	DISABLED	Sends BREAK level (space) on TxD

FIGURE 16.7. Transmitter control bits CR5 and CR6.

CR7	*Receive Interrupt*
0	DISABLED
1	ENABLED

FIGURE 16.8. Receive interrupt enable bit CR7.

A typical subroutine used to set up the ACIA is shown in Figure 16.9. Note that you must first execute a master reset and then set up the control register (CRSR) with the desired bit pattern.

```
ACIASU   LDA   #$03      ;Master reset
         STA   CRSR
         LDA   #$0D      ;7 bits, odd parity, 1 stop bit
         STA   CRSR
         RTS
```

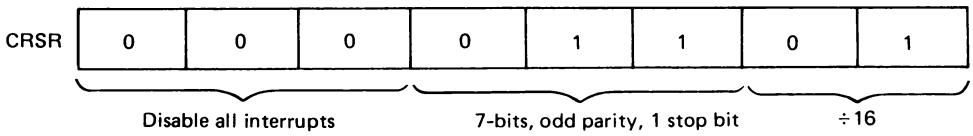


FIGURE 16.9. Typical ACIA setup subroutine.

The Status Register

The ACIA status register is shown in Figure 16.10. This is a read only register that shares the same address as the control register (CRSR).

Bit 0 of the status register is the receive data register full (RDRF) bit. This bit is set to 1 when the receive shift register has filled up with a

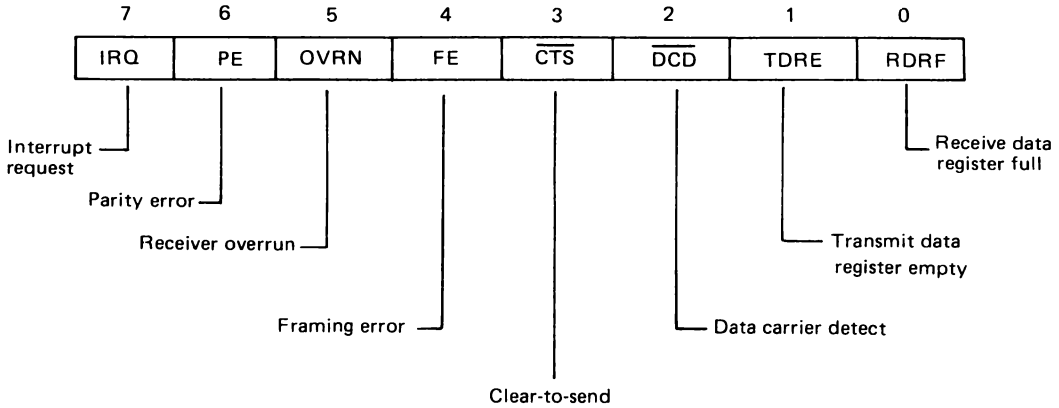


FIGURE 16.10. The ACIA status register.

complete byte and transferred this byte to the receive data register (RDR) (see Figure 16.3). The receive data register can then be read by the MPU.

Reading the receive data register will clear the RDRF bit in the status register (it is also cleared by a master reset). If bit 7 of the control register is set, enabling receive interrupts, then an interrupt will occur whenever the RDRF bit in the status register is set (indicating that the receive data register can be read). Alternatively, you can poll the RDRF bit to see if the receive data register is full. A typical polling sequence is shown in Figure 16.11. If the RDRF bit is set to 1, the receive data register (TDRRDR) is read. Otherwise the program branches to NEXT, which

might check the keyboard and then return to RECV to check the RDRF bit again. In Figure 16.11 the label CRSR is used for both the control register (CR) and the status register (SR) because they share the same address. Similarly, the label TDRRDR is used for both the transmit data register (TDR) and the receive data register (RDR) because they also share the same address.

```
RECV    LDA CRSR
        LSR             ;test RDRF
        BCC NEXT        ;if RDRF = 0, go to NEXT
        LDA TDRRDR      ;else read RDR
        —
        —
NEXT    —
```

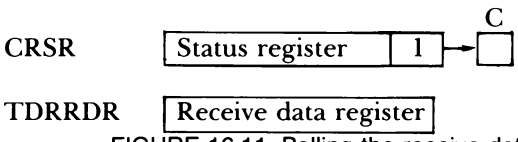


FIGURE 16.11. Polling the receive data register.

You must make sure that you check the RDRF bit often enough not to miss any incoming bytes. If the receive shift register is filled before the previous data in the receive data register have been read by the MPU, the overrun (OVRN) flag (bit 5) in the status register will be set to 1. If the receive interrupt enable bit (CR7) in the control register is set, the overrun condition will produce an interrupt. The OVRN flag is cleared by reading data from the receive data register or by a master reset.

All asynchronous serial data must end with at least 1 stop bit. If a stop bit does not occur where expected, a framing error is indicated and the FE flag (bit 4) is set to 1. This flag is set or reset after each byte is received. You must therefore test this bit between the time a character is received and the time the next character fills the receive shift register.

During the time a data character is in the receive data register, the parity error flag, PE (bit 6), in the status register will be set to 1 if the number of 1s in the byte does not agree with the preselected parity (odd or even). If 7 data bits plus a parity bit are selected, the parity bit (D7) will be set to 0 when the MPU reads the receive data register. This means that the program does not have to mask bit 7 after reading data from the ACIA.

Bit 1 of the status register is the transmit data register empty (TDRE) bit. This bit is set to 1 when data from the transmit data register (TDR) are transferred to the transmit shift register. These data are then shifted out through TxD at the baud rate, preceded by a start bit and

ending with a parity bit (if selected) and 1 or 2 stop bits (see Figure 16.3). While these data are being shifted out, another byte can be stored in the transmit data register. When this is done the TDRE bit is cleared to 0 and will remain 0 until the transmit shift register has finished shifting out the previous character. When the shift register is free to accept another byte, the contents of the TDR are transferred to the shift register and the TDRE flag (bit 1) in the status register is set to 1 again.

A subroutine that will send a byte of data stored in accumulator A is given in Figure 16.12. After saving the data on the stack, bit 1 (TDRE) of the status register is continually tested until it goes to 1. Then the data byte is retrieved from the stack and stored in the transmit data register. Another data byte can then by loaded into accumulator A and the SEND subroutine called again. It will wait for the previous character to be transferred to the shift register before storing the new byte in the transmit data register.

```
SEND    PHA             ;save data
SEND1   LDA CRSR
        LSR
        LSR             ;test TDRE
        BCC SEND1       ;wait for TDRE = 1
        PLA             ;restore data
        STA TDRRDR      ;Store data in TDR
        RTS
```

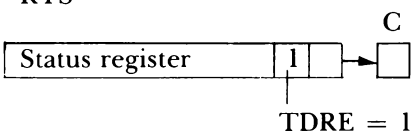


FIGURE 16.12. Subroutine to send a byte of data.

The clear-to-send ($\overline{\text{CTS}}$) and data carrier detect ($\overline{\text{DCD}}$) bits in the status register are associated with the use of a modem. A modem is used to modulate (and demodulate) a serial, digital signal so that it can be transmitted over telephone lines. The $\overline{\text{DCD}}$ bit will be set if the $\overline{\text{DCD}}$ line connected to a modem is high. This indicates that a carrier is not present at the modem and therefore valid data cannot be sent or received. When the $\overline{\text{DCD}}$ bit goes high, an interrupt will occur if the receive interrupts are enabled.

The $\overline{\text{CTS}}$ bit of the status register follows the $\overline{\text{CTS}}$ line from the

modem. This line must be low in order to transmit data. When this bit is high the transmit data register empty bit is 0.

Bit 7 (IRQ) of the status register will be set to 1 when the $\overline{\text{IRQ}}$ pin of the ACIA is low. This will occur when an enabled interrupt occurs. This bit is cleared by reading the receive data register or writing to the transmit data register.

AN APPLE II TERMINAL

The Apple II can be converted to a computer terminal by putting an ACIA on a peripheral board and connecting the transmit and receive lines TxD and RxD through RS232 buffers. A schematic for such a board is shown in Figure 16.13.*

The MC1488 chip is an RS232 transmitting buffer that converts a 0- to 5-volt logic signal to a $\pm$ 12-volt signal that is transmitted from a standard RS232 port. A logic 0 (0 volts) is converted to + 12 volts and a logic 1 (+ 5 volts) is converted to − 12 volts. These higher voltages give better noise immunity when sent over long distances.

The MC1489 chip is an RS232 receiving buffer that converts a $\pm$ 12-volt RS232 signal into a 0- to 5-volt logic signal that the ACIA can accept.

The MC14411 chip is a bit rate generator that can generate $\times$ 16 or $\times$ 64 clock frequencies corresponding to the common baud rates. It generates 16 different output frequencies that can be selected by a rotary switch. These clock signals are fed to the transmit and receive clock inputs on the ACIA.

A typical dumb terminal will check to see if a key has been pressed. If it has, it will send the character to the host computer. It will then check to see if the host computer is sending a character. If it is, it will display the character on the screen. Otherwise it will check the keyboard again.

The algorithm for a dumb terminal is given in Figure 16.14. This algorithm is for *full-duplex* communication, in which the host computer echoes each character that it receives. Therefore, the Apple II will not display a character typed on the Apple II keyboard until the character has been sent to the host computer and returned. The user will think that it is being displayed immediately, but actually it makes a round trip to the host computer. In half-duplex communication, the Apple II would display each character as it is typed and the host computer would not echo the characters it receives.

*See footnote associated with Figure 14.7 in Chapter 14.

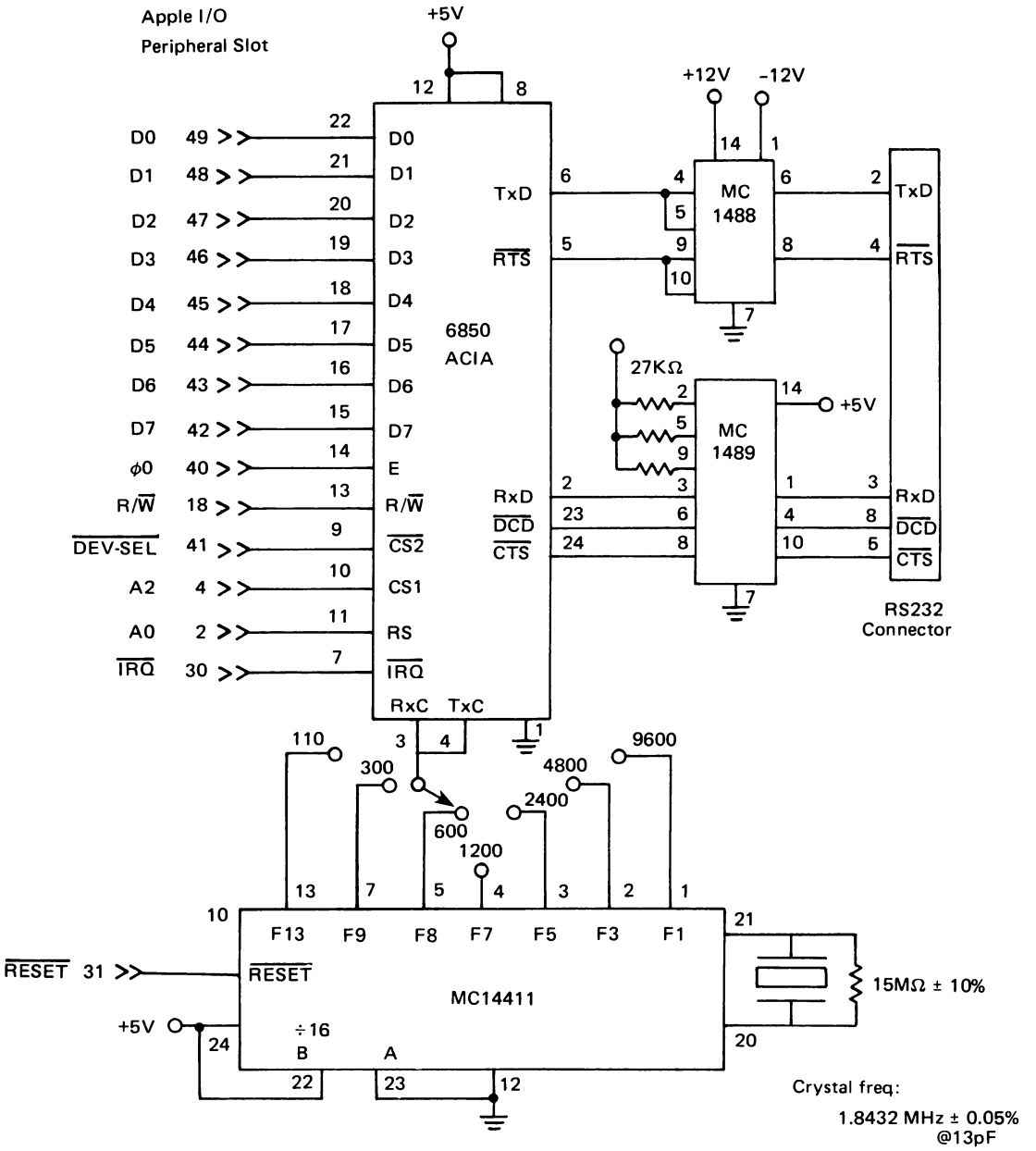


FIGURE 16.13. Schematic diagram of a serial I/O board for the Apple II.

```
loop:  if key has been pressed
       then send character to host computer
       if host computer has sent character
       then display character
repeat loop
```

FIGURE 16.14. Algorithm for a dumb terminal.

The assembly language program for this dumb terminal is shown in Figure 16.15. The program continually alternates between checking the keyboard and checking the ACIA. The subroutine SEND is the one shown in Figure 16.12. The built-in routine COUT1, which displays the character whose Apple ASCII code is in accumulator A, was described in Chapter 9. Note that bit 7 must be set to 1 to display normal characters on the screen.

```
COUT1   EQU   $FDF0
KEY     EQU   $C000
KEYSTB  EQU   $C010
CRSR    EQU   address of ACIA control/status register
TDRRDR  EQU   address of ACIA transmit/receive register

TERM    BIT   KEY       ;Is key pressed?
        BPL   RECV      ;if not, go to RECV
        BIT   KEYSTB    ;clear keyboard strobe
        LDA   KEY       ;read key value
        JSR   SEND      ;send to host computer
RECV    LDA   CRSR      ;check RDRF to see
        LSR             ;if host computer is sending
        BCC   TERM      ;if not, check keyboard
        LDA   TDRRDR    ;otherwise, read character
        ORA   #$80      ;set bit 7
        JSR   COUT1     ;display on terminal
        JMP   TERM      ;repeat again
```

FIGURE 16.15. Program for a dumb terminal.

EXERCISE 16.1

Write a printer routine that will send characters to a serial printer using an ACIA.

EXERCISE 16.2

Write a program that can be executed simultaneously on two different Apples connected by a serial line between two ACIAs. The program should be written so that any key pressed on either Apple will cause the character to be displayed on *both* Apples.

Appendices

APPENDIX A
6502 INSTRUCTION SET

Table A.1 Memory and Register Instructions

ADDRESSING MODES

	INHER/ACC	*IMMED*	*ZERO PG*	*ABSOLUTE*	*ZERO PG,X*	*ZERO PG,Y*
MNEMONIC	*OP ~ #*	*OP ~ #*	*OP ~ #*	*OP ~ #*	*OP ~ #*	*OP ~ #*
ADC		69 2 2	65 3 2	6D 4 3	75 4 2	
AND		29 2 2	25 3 2	2D 4 3	35 4 2	
ASL	0A 2 1		06 5 2	0E 6 3	16 6 2	
BIT			24 3 2	2C 4 3		
CMP		C9 2 2	C5 3 2	CD 4 3	D5 4 2	
CPX		E0 2 2	E4 3 2	EC 4 3		
CPY		C0 2 2	C4 3 2	CC 4 3		
DEC			C6 5 2	CE 6 3	D6 6 2	
DEX	CA 2 1					
DEY	88 2 1					
EOR		49 2 2	45 3 2	4D 4 3	55 4 2	
INC			E6 5 2	EE 6 3	F6 6 2	
INX	E8 2 1					
INY	C8 2 1					
LDA		A9 2 2	A5 3 2	AD 4 3	B5 4 2	
LDX		A2 2 2	A6 3 2	AE 4 3		B6 4 2
LDY		A0 2 2	A4 3 2	AC 4 3	B4 4 2	
LSR	4A 2 1		46 5 2	4E 6 3	56 6 2	
NOP	EA 2 1					
ORA		09 2 2	05 3 2	0D 4 3	15 4 2	
PHA	48 3 1					
PHP	08 3 1					
PLA	68 4 1					
PLP	28 4 1					
ROL	2A 2 1		26 5 2	2E 6 3	36 6 2	
ROR	6A 2 1		66 5 2	6E 6 3	76 6 2	
SBC		E9 2 2	E5 3 2	ED 4 3	F5 4 2	
STA			85 3 2	8D 4 3	95 4 2	
STX			86 3 2	8E 4 3		96 4 2
STY			84 3 2	8C 4 3	94 4 2	
TAX	AA 2 1					
TAY	A8 2 1					
TSX	BA 2 1					
TXA	8A 2 1					
TXS	9A 2 1					
TYA	98 2 1					

*Add 1 if page boundary is crossed
~ No. of cycles
# No. of bytes
Inherent (Inher)

ADDRESSING MODES				*CONDITION CODES*						
ABSOL,X	*ABSOL,Y*	*(IND,X)*	*(IND),Y*							
OP ~ #	*OP ~ #*	*OP ~ #*	*OP ~ #*	*MNEMONIC*	*N*	*V*	*D*	*I*	*Z*	*C*
7D 4* 3	79 4* 3	61 6 2	71 5* 2	ADC	√	√	-	-	√	√
3D 4* 3	39 4* 3	21 6 2	31 5* 2	AND	√	-	-	-	√	-
1E 7 3				ASL	√	-	-	-	√	√
				BIT	M_7	M_6	-	-	√	-
DD 4* 3	D9 4* 3	C1 6 2	D1 5* 2	CMP	√	-	-	-	√	√
				CPX	√	-	-	-	√	√
				CPY	√	-	-	-	√	√
DE 7 3				DEC	√	-	-	-	√	-
				DEX	√	-	-	-	√	-
				DEY	√	-	-	-	√	-
5D 4* 3	59 4* 3	41 6 2	51 5* 2	EOR	√	-	-	-	√	-
FE 7 3				INC	√	-	-	-	√	-
				INX	√	-	-	-	√	-
				INY	√	-	-	-	√	-
BD 4* 3	B9 4* 3	A1 6 2	B1 5* 2	LDA	√	-	-	-	√	-
	BE 4* 3			LDX	√	-	-	-	√	-
BC 4* 3				LDY	√	-	-	-	√	-
5E 7 3				LSR	0	-	-	-	√	√
				NOP	-	-	-	-	-	-
1D 4* 3	19 4* 3	01 6 2	11 5* 2	ORA	√	-	-	-	√	-
				PHA	-	-	-	-	-	-
				PHP	-	-	-	-	-	-
				PLA	√	-	-	-	√	-
				PLP	(RESTORED)					
3E 7 3				ROL	√	-	-	-	√	√
7E 7 3				ROR	√	-	-	-	√	√
FD 4* 3	F9 4* 3	E1 6 2	F1 5* 2	SBC	√	√	-	-	√	√
9D 5 3	99 5 3	81 6 2	91 6 2	STA	-	-	-	-	-	-
				STX	-	-	-	-	-	-
				STY	-	-	-	-	-	-
				TAX	√	-	-	-	√	-
				TAY	√	-	-	-	√	-
				TSX	√	-	-	-	√	-
				TXA	√	-	-	-	√	-
				TXS	-	-	-	-	-	-
				TYA	√	-	-	-	√	-

Table A.2 Branch and Jump Instructions

ADDRESSING MODES					
	Relative	*Absolute*	*Indirect*	*Inher*	*Condition Codes*
Mnemonic	*OP ~ #*	*OP ~ #*	*OP ~ #*	*OP ~ #*	*N V B D I Z C*
BCC	90 3* 2				UNCHANGED
BCS	B0 3* 2				
BEQ	F0 3* 2				
BMI	30 3* 2				
BNE	D0 3* 2				
BPL	10 3* 2				
BVC	50 3* 2				
BVS	70 3* 2				
JMP		4C 3 3	6C 5 3		
JSR		20 6 3			
RTS				60 6 1	

*Add 1 if branch occurs to different page

Table A.3 Interrupt and Status Register Instructions

ADDRESSING MODE								
	Inher	*Condition Codes*						
	OP ~ #	*N*	*V*	*B*	*D*	*I*	*Z*	*C*
BRK	00 7 1	-	-	1	-	-	-	-
CLC	18 2 1	-	-	-	-	-	-	0
CLD	D8 2 1	-	-	-	0	-	-	-
CLI	58 2 1	-	-	-	-	0	-	-
CLV	B8 2 1	-	0	-	-	-	-	-
RTI	40 6 1	(RESTORED)						
SEC	38 2 1	-	-	-	-	-	-	1
SED	F8 2 1	-	-	-	1	-	-	-
SEI	78 2 1	-	-	-	-	1	-	-

APPENDIX B
THE TUTOR MONITOR

The TUTOR monitor will run on an Apple II or Apple II Plus with 48K bytes of RAM (random access memory). The program will automatically start executing when you boot the disk (either by turning on an Apple II Plus with the disk in the disk drive or by typing PR#6). The disk will boot on a 3.3-DOS (disk operating system). The TUTOR monitor occupies memory between the hex addresses $8000 and $93B2. It also uses some zero page addresses between $00C0 and $00FF. This means, among other things, that Applesoft BASIC will not work properly once the TUTOR monitor has been run until you re-boot the system. You can, however, use the Apple II monitor from the TUTOR monitor. In addition, DOS commands can be executed from within the TUTOR monitor.

Because the TUTOR monitor allows you to look anywhere in memory and change any value you want, you may sometimes inadvertently "bomb" the system. When this happens, pressing the RESET key will usually cause the TUTOR monitor to be executed again from the beginning. If this fails, you will have to turn off the computer and re-boot the system.

A summary of the TUTOR monitor commands is given at the end of this appendix. The basic operation of the TUTOR monitor is introduced in Chapters 2 and 3. The following commands are described, with examples, at the indicated pages in the book:

Command

/B Set breakpoint, page 67
/D Delete a block of bytes, page 100
/E Execute a program, page 68
/I Insert any number of hex bytes, page 99
/L List a disassembled portion of memory, page 94
/M Enter hex or ASCII values in memory, page 22
/O Calculate branching offset, page 53
/P Print a disassembled portion of memory on a printer, page 172
/R Change the contents of a register, page 13

The following TUTOR commands are not described elsewhere in the book:

/F Find a particular string of bytes.
After the prompt

FIND: ENTER HEX VALUES

enter any number of 2-digit hex values. When you press RETURN, the cursor will move to the first byte in the sequence.

/S Save or load a binary file on disk.
After the prompt

STORAGE: L S

press either L to load a file from disk, or S to save a file on disk.
If you press S the prompt

BSAVE NAME,A$----,L$----

will appear on the command line. Type this on the entry line substituting your file name for NAME, the starting address of your program following A$, and the length, in bytes, of your program following L$. For example, if you type

BSAVE PROGRAM 5,A$300,L$3B

then the $3B bytes between $300 and $33A will be stored on disk as the binary file PROGRAM 5.
After pressing /SL the prompt

BLOAD NAME

will appear on the command line. Type this on the entry line substituting your file name for NAME. For example, if you type

BLOAD PROGRAM 5

the binary file PROGRAM 5 will be loaded into memory at the starting address at which it was BSAVEd. This starting address will be displayed in the PC register on the TUTOR monitor screen.

/T Transfer a block of bytes.
In response to the prompt

TRANSFER: DESTINATION ADDRESS

enter the address to which the first byte in the block is to be moved. After pressing RETURN the prompt

TRANSFER: NO. OF BYTES

will appear on the command line. Enter the number of bytes to be moved. When you press RETURN, the block of bytes will be transferred to the new location and the cursor will move to the first byte of this new block. Any number of bytes can be moved either forward or backward in memory. If you try to move data into some parts of memory, such as that occupied by the TUTOR monitor or certain I/O addresses, the computer will "crash" and you will have to reload the TUTOR monitor.

/Q Quit to the Apple monitor or to DOS.
In response to the prompt

QUIT: D M

pressing M will clear the screen and put you in the Apple monitor with the * prompt. The use of this monitor is described in the *Apple II Reference Manual.*

Pressing /QD will display the prompt

ENTER DOS COMMAND

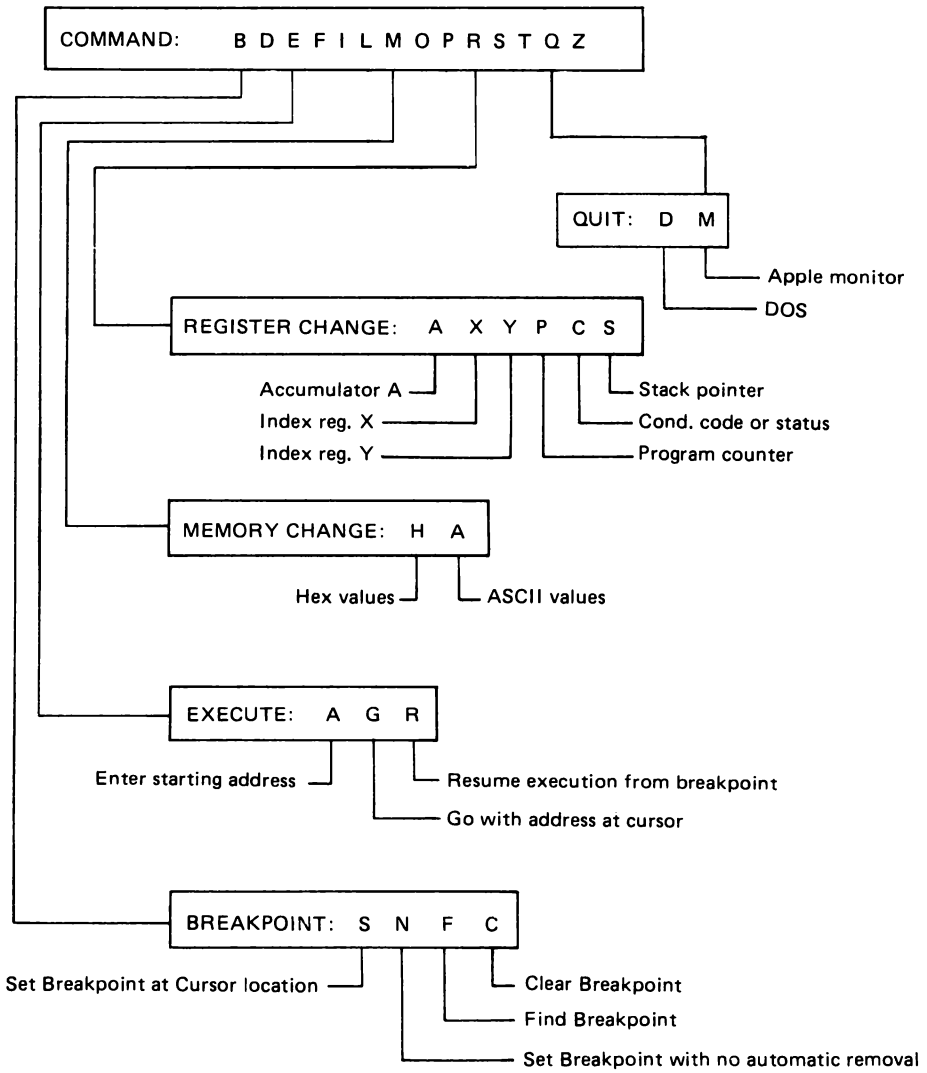
on the command line with the prompt ! on the entry line. You can now enter any DOS command such as CATALOG. After executing the DOS command, pressing any key will return to the TUTOR monitor.

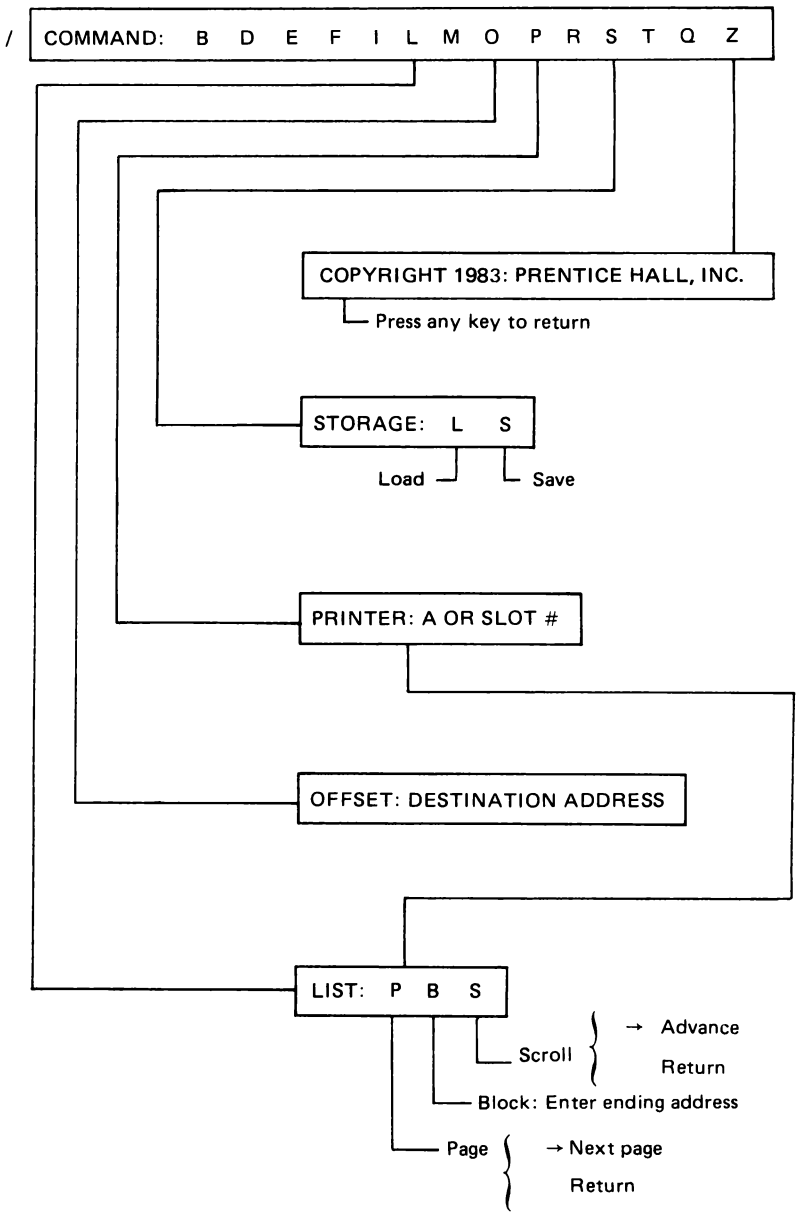
/Z will display the copyright message

COPYRIGHT 1983: PRENTICE-HALL, INC.

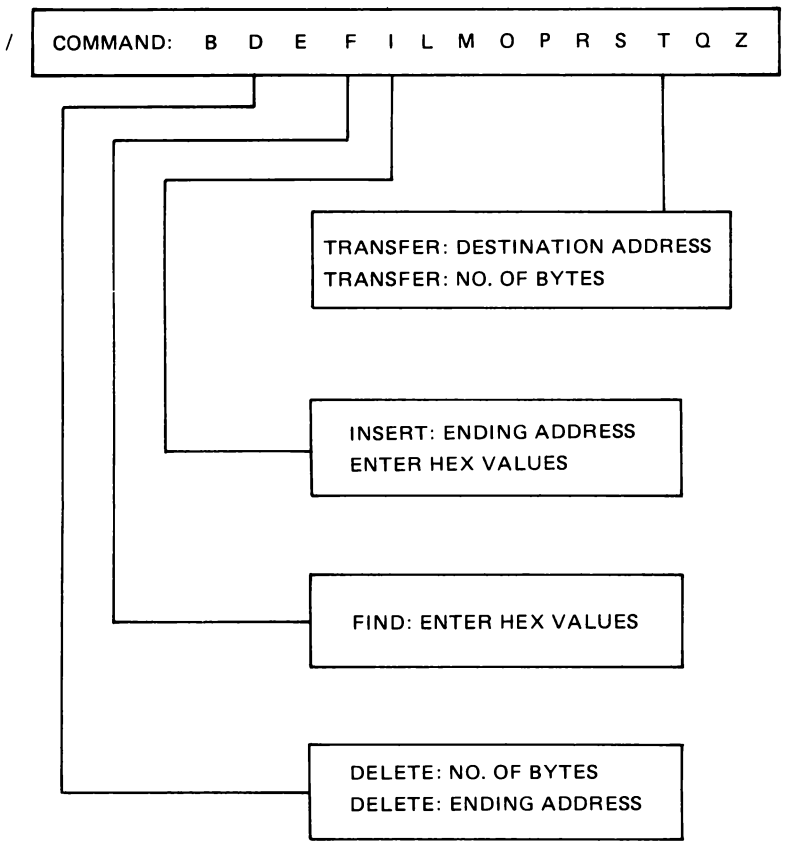
on the command line. Pressing any key will return to the TUTOR monitor.

→ advance byte (—); Advance row (:)
← Back up 1 byte (—); Back up 1 row (:)
space bar: toggle (—) (:)
> Go To: MEMORY ADDRESS
CTRL S: Single step instruction at >




/ COMMAND: B D E F I L M O P R S T Q Z
COPYRIGHT 1983: PRENTICE HALL, INC.
Press any key to return
STORAGE: L S
Load
Save
PRINTER: A OR SLOT #
OFFSET: DESTINATION ADDRESS
LIST: P B S
Scroll
→ Advance
Return
Block: Enter ending address
Page
→ Next page
Return

6502 TUTOR MONITOR SUMMARY



APPENDIX C
USING MACHINE LANGUAGE SUBROUTINES WITH BASIC

If you have a machine language subroutine there are two ways that you can access this subroutine from a BASIC program. The first is to use the CALL command, and the second is to use the USR function.

The BASIC command CALL *Addr* will transfer control to a machine language program starting at the decimal address *Addr*. This command can be used in either the immediate mode or the deferred mode. When the machine language subroutine executes an RTS instruction, control will return to the calling BASIC program.

Data values can be passed to and from a machine language subroutine by using the USR function. When the BASIC statement X=USR(A) is executed the value of A is placed in the floating point accumulator in the hex locations $9D-$A3, and control is then transferred to location $000A where a JMP instruction to your machine language program must be executed. Thus, $4C (JMP) must be placed in location $000A and the starting address of your machine language subroutine must be placed in locations $000B (LSB) and $000C (MSB).

Within the machine language subroutine the floating point value of A can be converted to a 16-bit integer value by executing a built-in floating-to-integer subroutine at location $E10C using JSR $E10C. After executing this subroutine the high order byte of A is in location $00A0 and the low order byte is in location $00A1. This integer value can then be used in your machine language subroutine.

When your machine language subroutine executes an RTS instruction, the floating point number currently stored in the floating point accumulator will be assigned to the function USR. That is, it will become the value of X in the statement X=USR(A). In order to pass a 16-bit integer value, V, back to the BASIC program, store the most significant byte in accumulator A, the least significant byte in index register Y, and execute JSR $E2F2 which is a built-in integer-to-floating subroutine. Then execute an RTS instruction which will return control to the BASIC program with the value of USR equal to the original 16-bit integer value V.

MOTOROLA
SEMICONDUCTORS
3501 ED BLUESTEIN BLVD., AUSTIN, TEXAS 78721

MC14443
MC14447

Advance Information

CMOS MSI
(LOW-POWER COMPLEMENTARY MOS)

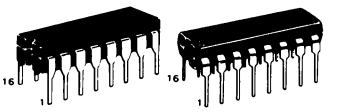
MICROPROCESSOR BASED ANALOG TO DIGITAL CONVERTER

ANALOG TO DIGITAL CONVERTER LINEAR SUBSYSTEM

The MC14443 and the MC14447 devices are 6 channel, single slope, 8-10 bit analog to digital converter linear subsystems for microprocessor based data and control, systems. Contained in both devices are a one of 8 decoder, an 8 channel analog multiplexer, a buffer amplifier, a precision voltage to current converter, a ramp start circuit, and a comparator. The output driver of the MC14443 comparator is an open-drain N-channel capable of sinking up to 5 mA of current. The output of the MC14447 comparator has a standard B-Series P-channel, N-channel pair.

A processor system (such as the MC6800, MC141000 or MC3870) provides the addressing, timing, counting, and arithmetic operations required for implementing a full analog to digital converter system. A system made up of a processor and the linear subsystem has features such as automatic zeroing and variable scaling (weighting) of six separate analog channels.

- Quiescent Current 0.8 mA Typical at V_{DD} = 5 V
- Single Supply Operation +4.5 to +18 Volts
- MPU Compatible
- Typical Resolution — 10 Bits
- Typical Conversion Cycle as Fast as 300 μs
- Ratio Metric Conversion Minimizes Error



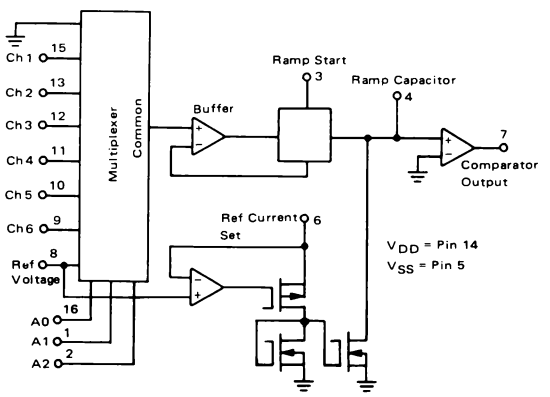
L SUFFIX
CERAMIC PACKAGE
CASE 620

P SUFFIX
PLASTIC PACKAGE
CASE 648

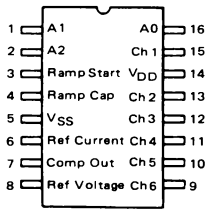
ORDERING INFORMATION

MC14XXX — Suffix Denotes
- L Ceramic Package
- P Plastic Package
- A Extended Operating Temperature Range
- C Limited Operating Temperature Range

BLOCK DIAGRAM



PIN ASSIGNMENT



This is advance information and specifications are subject to change without notice.

ADI-476

MAXIMUM RATINGS (Voltages referenced to V_{SS})

Rating	Symbol	Value	Unit
DC Supply Voltage	V_{DD}	-0.5 to +18	Vdc
Input Voltage, All Inputs	V_{in}	-0.5 to V_{DD} + 0.5	Vdc
DC Current Drain per Pin	I	10	mAdc
Operating Temperature Range – AL Device CL/CP Device	T_A	-55 to +125 -40 to +85	°C
Storage Temperature Range	T_{stg}	-65 to +150	°C

This device contains circuitry to protect the inputs against damage due to high static voltages or electric fields; however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum rated voltages to this high impedance circuit. For proper operation it is recommended that V_{in} and V_{out} be constrained to the range $V_{SS} \leq (V_{in}$ or $V_{out}) \leq V_{DD}$.

ELECTRICAL CHARACTERISTICS

Characteristic	Symbol	V_{DD} Vdc	T_{low}* Min	T_{low}* Max	25°C Min	25°C Typ	25°C Max	T_{high}* Min	T_{high}* Max	Unit
Output Voltage—Comparator "0" Level	V_{OL}	5.0	–	0.05	–	0.01	0.05	–	0.05	Vdc
V_{in} @ Pin 4 = 0 V		10	–	0.05	–	0.01	0.05	–	0.05	
		15	–	0.05	–	0.01	0.05	–	0.05	
V_{in} @ Pin 4 = 0.5 V "1" Level	V_{OH}	5.0	4.95	–	4.95	4.99	–	4.95	–	Vdc
(R_L = 10 k, MC14447 only)		10	9.95	–	9.95	9.99	–	9.95	–	
		15	14.95	–	14.95	14.99	–	14.95	–	
Input Voltage-Address, Ramp Start "0 Level"	V_{IL}									Vdc
(V_O = 4.5 or 0.5 Vdc)		5.0	–	1.5	–	2.25	1.5	–	1.5	
(V_O = 9.0 or 1.0 Vdc)		10	–	3.0	–	4.50	3.0	–	3.0	
(V_O = 13.5 or 1.5 Vdc)		15	–	4.0	–	6.75	4.0	–	4.0	
"1" Level	V_{IH}									Vdc
(V_O = 0.5 or 4.5 Vdc)		5.0	3.5	–	3.5	2.75	–	3.5	–	
(V_O = 1.0 or 9.0 Vdc)		10	7.0	–	7.0	5.50	–	7.0	–	
(V_O = 1.5 or 13.5 Vdc)		15	11.0	–	11.0	8.25	–	11.0	–	
Output Drive Current—Comparator	I_{OH}									mAdc
V_{in} @ Pin 4 = 0.5 V (MC14447 only)										
(V_{OH} = 2.5 Vdc)		5.0	-2.5	–	-2.1	-4.2	–	-1.7	–	
(V_{OH} = 4.6 Vdc)		5.0	-0.52	–	-0.44	-0.88	–	-0.36	–	
(V_{OH} = 9.5 Vdc)		10	-1.3	–	-1.1	-2.25	–	-0.9	–	
(V_{OH} = 13.5 Vdc)		15	-3.6	–	-3.0	-8.8	–	-2.4	–	
V_{in} @ Pin 4 = 0 V	I_{OL}									mAdc
(V_{OL} = 0.4 Vdc)		5.0	0.52	–	0.44	0.88	–	0.36	–	
(V_{OL} = 0.5 Vdc)		10	1.3	–	1.1	2.25	–	0.9	–	
(V_{OL} = 1.5 Vdc)		15	3.6	–	3.0	8.8	–	2.4	–	
Input Current—Address, Ramp Start	I_{in}	15	–	±0.3	–	–	±0.3	–	±1.0	μAdc
Input Current—Analog Inputs	I_{in}	15	–	–	–	±0.1	±10	–	–	nAdc
Input Capacitance—Address, Ramp Start V_{in} = 0 V	C_{in}	15	–	–	–	5.0	7.5	–	–	pF
Quiescent Current	I_{DD}	5	–	–	–	0.8	–	–	–	mAdc
		10	–	–	–	1.5	–	–	–	
		15	–	–	–	1.7	–	–	–	
Crosstalk Between Any Two Input Channels	V_{Cr}	–	–	–	–	0	4.0	–	–	mVdc
Reference Current Range	I_R	–	–	–	10	–	50	–	–	μAdc
Common Mode Input Voltage	V_{CM}	5	–	–	0	–	2.5	–	–	Vdx
		10	–	–	0	–	7.0	–	–	
		15	–	–	0	–	12	–	–	
Buffer Amplifier Output Offset	V_{BO}	5	–	–	–	0.285	–	–	–	Vdc
		10	–	–	–	0.400	–	–	–	
		15	–	–	–	0.420	–	–	–	
Comparator Threshold	V_{TC}	5	–	–	–	0.195	V_{BO}	–	–	Vdc
		10	–	–	–	0.275	V_{BO}	–	–	
		15	–	–	–	0.290	V_{BO}	–	–	
Reference Voltage Range	V_R	5	–	–	2.0	–	2.5	–	–	Vdc
		10	–	–	2.0	–	7	–	–	
		15	–	–	2.0	–	12	–	–	
Conversion Linearity V_{in} = V_{DD} – 3 V for C > 100 pF	L_C	–	–	–	–	0.05	0.2	–	–	% Full Scale

*T_{low} = -55°C for AL Device, -40°C for CL/CP Device.
T_{high} = +125°C for AL Device, +85°C for CL/CP Device.

SWITCHING CHARACTERISTICS (C_L = 50 pF, T_A = 25°C)

Characteristic		Symbol	V_{DD} Vdc	Typ	Max	Unit
Output Rise Time—Comparator	(MC14447 only)	t_r	5.0 10 15	100 50 40	200 100 80	ns
Output Fall Time—Comparator		t_f	5.0 10 15	40 20 15	80 40 30	ns
Propagation Delay Time—Comparator	MC14443 (R_L = 10 k to V_{DD})	t_{PLH}	5.0 10 15	2.0 1.0 0.75	– – –	μs
	MC14447	t_{PLH}	5.0 10 15	2.0 1.0 0.75	– – –	μs
	All Devices	t_{PHL}	5.0 10 15	2.0 1.0 0.75	– – –	μs
Multiplexer Propagation Delay		t_M	5.0 10 15	850 300 250	– – –	ns
Ramp Start Delay Time		t_{RS}	5.0 10 15	850 300 250	– – –	ns
Acquisition Time*		t_A	5.0 10 15	10 9.0 8.0	– – –	μs

*Acquisition Time includes multiplexer propagation delay, ramp start propagation delay and the time required to charge ramp capacitor to the selected input voltage.

DEVICE OPERATION

ADDRESS INPUTS SELECT (A0, A1, A2, Pins 1, 2, 16) The input voltage source to be presented to the measurement system according to the Truth Table shown in Figure 3.

RAMP START (Ramp Start, Pin 3) When the Ramp Start is low, the ramp capacitor is charged to a voltage associated with the selected input channel. When the Ramp Start is brought high,the connection to the input channel is broken and the capacitor begins to ramp toward V_{SS}.

RAMP CAPACITOR (Ramp Cap, Pin 4) The ramp capacitor is used to generate a time period when discharged from a selected voltage via a precise reference current.

NEGATIVE POWER SUPPLY (V_{SS}, Pin 5) This pin is system ground.

REFERENCE CURRENT (Ref Current, Pin 6) To discharge the ramp capacitor, the reference current is fixed via a resistor (R_{Ref}) to a positive supply from pin 6. Typical current is equal to $(V_{DD} - V_{Ref})/R_{Ref}$.

COMPARATOR OUTPUT (Comp Out, Pin 7) This output is low when the capacitor has reached the discharged voltage and is high otherwise.

REFERENCE VOLTAGE (Ref Voltage, Pin 8) This voltage can be set to a voltage between V_{SS} + 2 V and V_{DD} – 2 V. This is the known voltage to which the unknown is compared.

INPUT CHANNELS (Pins 9, 10, 11, 12, 13, 15) Input channels 1 through 6 are used to monitor up to six separate unknown voltages. Selection is via the address inputs.

POSITIVE POWER SUPPLY (V_{DD}, Pin 14) This pin is the package positive power supply pin.

FIGURE 1 – VOLTAGE TO PULSE WIDTH CONVERSION

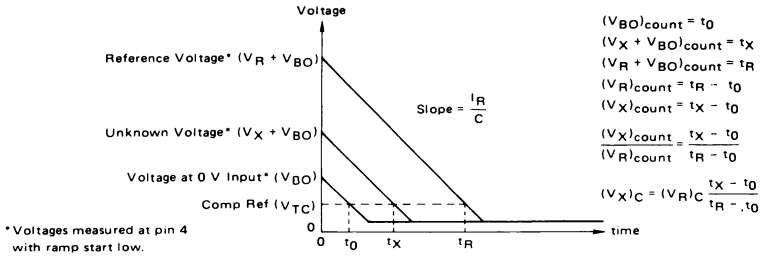
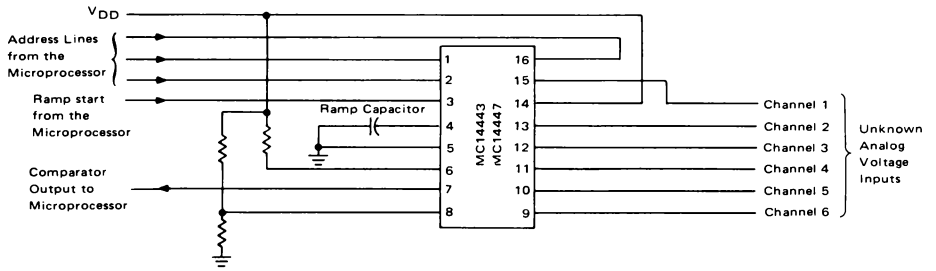


FIGURE 2 – TRUTH TABLE

A2	A1	A0	Input Selected	
0	0	0	V_{SS}	Channel 0 (ground)
0	0	1	Ch1	Channel 1
0	1	0	Ch2	Channel 2
0	1	1	Ch3	Channel 3
1	0	0	Ch4	Channel 4
1	0	1	Ch5	Channel 5
1	1	0	Ch6	Channel 6
1	1	1	V_{Ref}	Channel 7 (External Reference)

FIGURE 3 – TYPICAL APPLICATIONS CIRCUIT



CONVERSION SEQUENCE

Step No.	A2	A1	A3	Ramp Start	Comment
1.	1	1	1	0	Channel 7 Selected (Reference Voltage)
2.	1	1	1	1	Record time until Pin 7 goes low
3.	0	0	0	0	Channel 0 Selected (Ground)
4.	0	0	0	1	Record time until Pin 7 goes low
5.	0	0	1	0	Channel 1 Selected
6.	0	0	1	1	Record time until Pin 7 goes low
Calculate $t_{Ch7} - t_{Ch0} = t_{Ch7'}$ Step 2 – Step 4					
Calculate $t_{Ch1} - t_{Ch0} = t_{Ch1'}$ Step 6 – Step 4					
Calculate $V_{unknown} = V_{Ch7}(t_{Ch1'}/t_{Ch7'})$					
7.	0		0		Channel 2 Selected
8.	0		1		Record time until Pin 7 goes low
Calculate $t_{Ch2} - t_{Ch0} = t_{Ch2'}$					
Calculate $V_{unknown} = V_{Ch7}(t_{Ch2'}/t_{Ch7'})$					
etc.					

MOTOROLA *Semiconductor Products Inc.*
3501 ED BLUESTEIN BLVD., AUSTIN, TEXAS 78721 • A SUBSIDIARY OF MOTOROLA INC.

MOTOROLA Semiconductors
3501 ED BLUESTEIN BLVD. AUSTIN, TEXAS 78721

MC6821

PERIPHERAL INTERFACE ADAPTER (PIA)

The MC6821 Peripheral Interface Adapter provides the universal means of interfacing peripheral equipment to the MC6800 Microprocessing Unit (MPU). This device is capable of interfacing the MPU to peripherals through two 8-bit bidirectional peripheral data buses and four control lines. No external logic is required for interfacing to most peripheral devices.

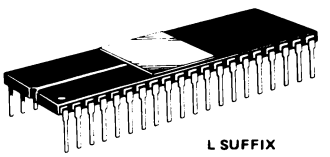
The functional configuration of the PIA is programmed by the MPU during system initialization. Each of the peripheral data lines can be programmed to act as an input or output, and each of the four control/interrupt lines may be programmed for one of several control modes. This allows a high degree of flexibility in the over-all operation of the interface.

- 8-Bit Bidirectional Data Bus for Communication with the MPU
- Two Bidirectional 8-Bit Buses for Interface to Peripherals
- Two Programmable Control Registers
- Two Programmable Data Direction Registers
- Four Individually-Controlled Interrupt Input Lines; Two Usable as Peripheral Control Outputs
- Handshake Control Logic for Input and Output Peripheral Operation
- High-Impedance 3-State and Direct Transistor Drive Peripheral Lines
- Program Controlled Interrupt and Interrupt Disable Capability
- CMOS Drive Capability on Side A Peripheral Lines
- Two TTL Drive Capability on All A and B Side Buffers
- TTL-Compatible
- Static Operation

MOS

(N-CHANNEL, SILICON-GATE, DEPLETION LOAD)

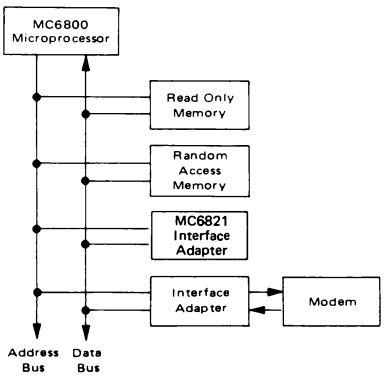
PERIPHERAL INTERFACE ADAPTER



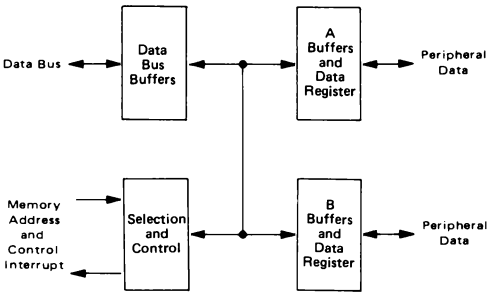
L SUFFIX
CERAMIC PACKAGE
CASE 715

NOT SHOWN: **P SUFFIX**
PLASTIC PACKAGE
CASE 711

M6800 MICROCOMPUTER FAMILY BLOCK DIAGRAM



MC6821 PERIPHERAL INTERFACE ADAPTER BLOCK DIAGRAM



DS 9435

ELECTRICAL CHARACTERISTICS (V_{CC} = 5.0 V ±5%, V_{SS} = 0, T_A = 0 to 70°C unless otherwise noted.)

Characteristic	Symbol	Min	Typ	Max	Unit
Input High Voltage	V_{IH}	V_{SS} + 2.0	–	V_{CC}	Vdc
Input Low Voltage	V_{IL}	V_{SS} – 0.3	–	V_{SS} + 0.8	Vdc
Input Leakage Current R/W, $\overline{\text{Reset}}$, RS0, RS1, CS0, CS1, $\overline{\text{CS2}}$, CA1, (V_{in} = 0 to 5.25 Vdc) CB1, Enable	I_{in}	–	1.0	2.5	μAdc
Three-State (Off State) Input Current D0–D7, PB0–PB7, CB2 (V_{in} = 0.4 to 2.4 Vdc)	I_{TSI}	–	2.0	10	μAdc
Input High Current PA0–PA7, CA2 (V_{IH} = 2.4 Vdc)	I_{IH}	-200	-400	–	μAdc
Input Low Current PA0–PA7, CA2 (V_{IL} = 0.4 Vdc)	I_{IL}	–	-1.3	-2.4	mAdc
Output High Voltage	V_{OH}				Vdc
(I_{Load} = -205 μAdc) D0–D7		V_{SS} + 2.4	–	–	
(I_{Load} = -200 μAdc) Other Outputs		V_{SS} + 2.4	–	–	
Output Low Voltage	V_{OL}				Vdc
(I_{Load} = 1.6 mAdc) D0–D7		–	–	V_{SS} + 0.4	
(I_{Load} = 3.2 mAdc) Other Outputs		–	–	V_{SS} + 0.4	
Output Leakage Current (Off State) $\overline{\text{IRQA}}$, $\overline{\text{IRQB}}$ (V_{OH} = 2.4 Vdc)	I_{LOH}	–	1.0	10	μAdc
Power Dissipation	P_D	–	–	550	mW
Capacitance	C_{in}				pF
(V_{in} = 0, T_A = 25°C, f = 1.0 MHz) D0–D7		–	–	12.5	
PA0–PA7, PB0–PB7, CA2, CB2		–	–	10	
Enable, R/W, $\overline{\text{Reset}}$, RS0, RS1, CS0, CS1, $\overline{\text{CS2}}$, CA1, CB1		–	–	7.5	
$\overline{\text{IRQA}}$, $\overline{\text{IRQB}}$	C_{out}	–	–	5.0	pF
Peripheral Data Setup Time (Figure 1)	t_{PDSU}	200	–	–	ns
Peripheral Data Hold Time (Figure 1)	t_{PDH}	0	–	–	ns
Delay Time, Enable negative transition to CA2 negative transition (Figure 2, 3)	t_{CA2}	–	–	1.0	μs
Delay Time, Enable negative transition to CA2 positive transition (Figure 2)	t_{RS1}	–	–	1.0	μs
Rise and Fall Times for CA1 and CA2 input signals (Figure 3)	t_r, t_f	–	–	1.0	μs
Delay Time from CA1 active transition to CA2 positive transition (Figure 3)	t_{RS2}	–	–	2.0	μs
Delay Time, Enable negative transition to Peripheral Data Valid (Figures 4, 5)	t_{PDW}	–	–	1.0	μs
Delay Time, Enable negative transition to Peripheral CMOS Data Valid (V_{CC} - 30% V_{CC}, Figure 4; Figure 12 Load C) PA0–PA7, CA2	t_{CMOS}	–	–	2.0	μs
Delay Time, Enable positive transition to CB2 negative transition (Figure 6, 7)	t_{CB2}	–	–	1.0	μs
Delay Time, Peripheral Data Valid to CB2 negative transition (Figure 5)	t_{DC}	20	–	–	ns
Delay Time, Enable positive transition to CB2 positive transition (Figure 6)	t_{RS1}	–	–	1.0	μs
Peripheral Control Output Pulse Width, CA2/CB2 (Figures 2, 6)	PW_{CT}	550	–	–	ns
Rise and Fall Time for CB1 and CB2 input signals (Figure 7)	t_r, t_f	–	–	1.0	μs
Delay Time, CB1 active transition to CB2 positive transition (Figure 7)	t_{RS2}	–	–	2.0	μs
Interrupt Release Time, $\overline{\text{IRQA}}$ and $\overline{\text{IRQB}}$ (Figure 9)	t_{IR}	–	–	1.6	μs
Interrupt Response Time (Figure 8)	t_{RS3}	–	–	1.0	μs
Interrupt Input Pulse Width (Figure 8)	PW_I	500	–	–	ns
Reset Low Time* (Figure 10)	t_{RL}	1.0	–	–	μs

*The Reset line must be high a minimum or 1.0 μs before addressing the PIA.

MAXIMUM RATINGS

Rating	Symbol	Value	Unit
Supply Voltage	V_{CC}	−0.3 to +7.0	Vdc
Input Voltage	V_{in}	−0.3 to +7.0	Vdc
Operating Temperature Range	T_A	0 to +70	°C
Storage Temperature Range	T_{stg}	−55 to +150	°C
Thermal Resistance	θ_{JA}	82.5	°C/W

This device contains circuitry to protect the inputs against damage due to high static voltages or electric fields; however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum rated voltages to this high impedance circuit.

BUS TIMING CHARACTERISTICS

READ (Figures 11 and 13)

Characteristic	Symbol	Min	Typ	Max	Unit
Enable Cycle Time	t_{cycE}	1.0	–	–	μs
Enable Pulse Width, High	PW_{EH}	0.45	–	–	μs
Enable Pulse Width, Low	PW_{EL}	0.43	–	–	μs
Setup Time, Address and R/W valid to Enable positive transition	t_{AS}	160	–	–	ns
Data Delay Time	t_{DDR}	–	–	320	ns
Data Hold Time	t_H	10	–	–	ns
Address Hold Time	t_{AH}	10	–	–	ns
Rise and Fall Time for Enable input	t_{Er}, t_{Ef}	–	–	25	ns

WRITE (Figures 12 and 13)

Enable Cycle Time	t_{cycE}	1.0	–	–	μs
Enable Pulse Width, High	PW_{EH}	0.45	–	–	μs
Enable Pulse Width, Low	PW_{EL}	0.43	–	–	μs
Setup Time, Address and R/W valid to Enable positive transition	t_{AS}	160	–	–	ns
Data Setup Time	t_{DSW}	195	–	–	ns
Data Hold Time	t_H	10	–	–	ns
Address Hold Time	t_{AH}	10	–	–	ns
Rise and Fall Time for Enable input	t_{Er}, t_{Ef}	–	–	25	ns

FIGURE 1 – PERIPHERAL DATA SETUP AND HOLD TIMES
(Read Mode)

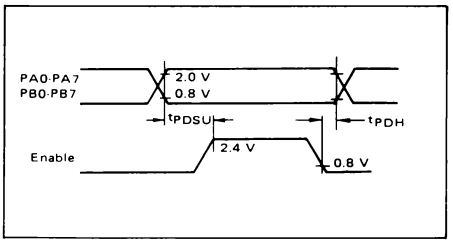


FIGURE 2 – CA2 DELAY TIME
(Read Mode; CRA-5 = CRA-3 = 1, CRA-4 = 0)

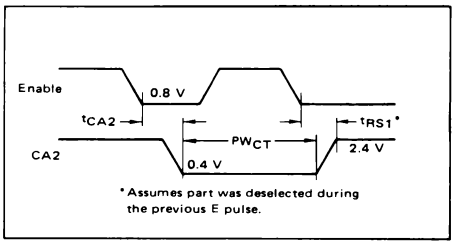


FIGURE 3 – CA2 DELAY TIME
(Read Mode; CRA-5 = 1, CRA-3 = CRA-4 = 0)

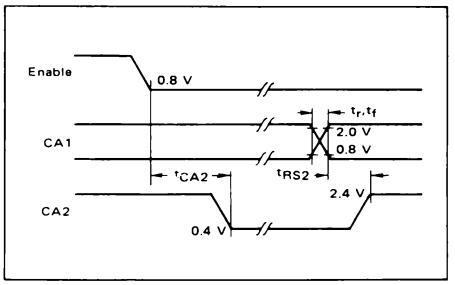


FIGURE 4 – PERIPHERAL CMOS DATA DELAY TIMES
(Write Mode; CRA-5 = CRA-3 = 1, CRA-4 = 0)

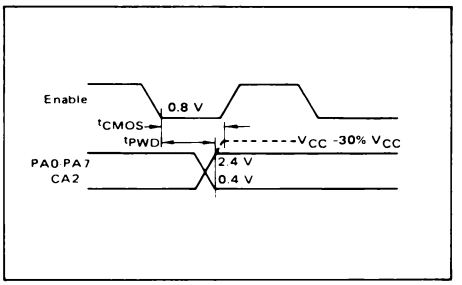


FIGURE 5 – PERIPHERAL DATA AND CB2 DELAY TIMES
(Write Mode; CRB-5 = CRB-3 = 1, CRB-4 = 0)

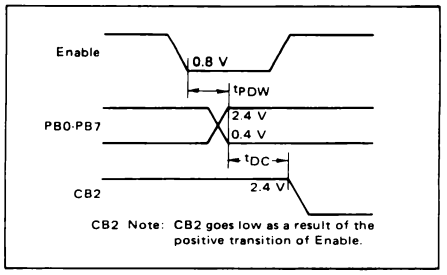


FIGURE 6 – CB2 DELAY TIME
(Write Mode; CRB-5 = CRB-3 = 1, CRB-4 = 0)

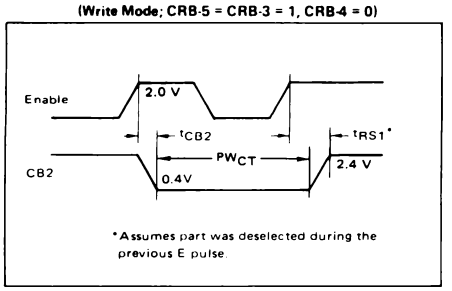


FIGURE 7 – CB2 DELAY TIME
(Write Mode; CRB-5 = 1, CRB-3 = CRB-4 = 0)

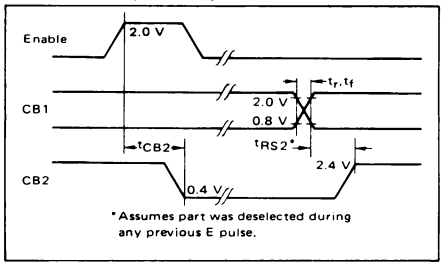


FIGURE 8 – INTERRUPT PULSE WIDTH and $\overline{IRQ}$ RESPONSE

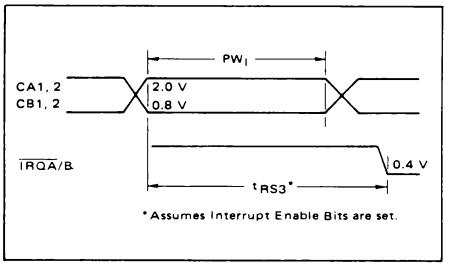


FIGURE 9 – $\overline{IRQ}$ RELEASE TIME

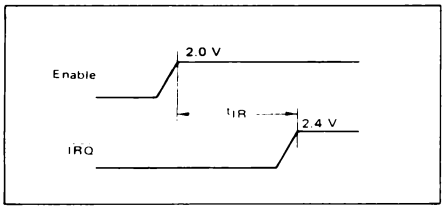


FIGURE 10 – $\overline{RESET}$ LOW TIME

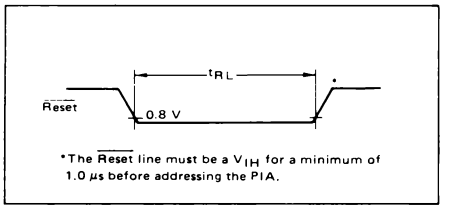


FIGURE 11 – BUS READ TIMING CHARACTERISTICS
(Read Information from PIA)

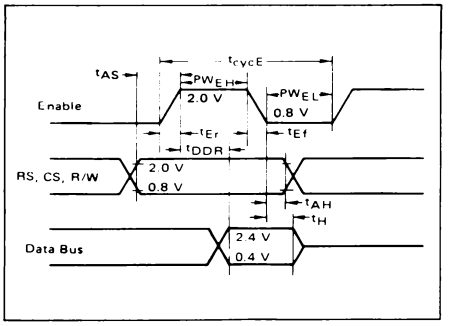


FIGURE 12 – BUS WRITE TIMING CHARACTERISTICS
(Write Information into PIA)

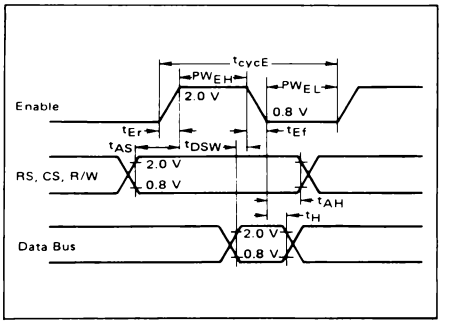
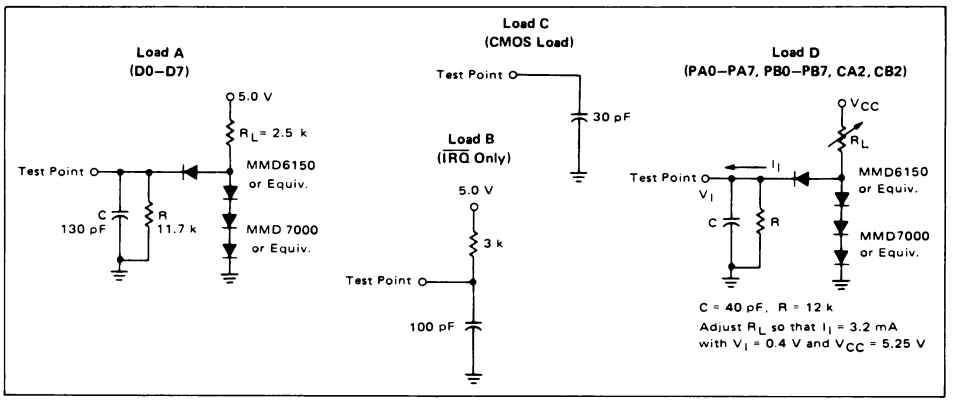


FIGURE 13 – BUS TIMING TEST LOADS



PIA INTERFACE SIGNALS FOR MPU

The PIA interfaces to the MC6800 MPU with an eight-bit bi-directional data bus, three chip select lines, two register select lines, two interrupt request lines, read/write line, enable line and reset line. These signals, in conjunction with the MC6800 VMA output, permit the MPU to have complete control over the PIA. VMA should be utilized in conjunction with an MPU address line into a chip select of the PIA.

PIA Bi-Directional Data (D0-D7) – The bi-directional data lines (D0-D7) allow the transfer of data between the MPU and the PIA. The data bus output drivers are three-state devices that remain in the high-impedance (off) state except when the MPU performs a PIA read operation. The Read/Write line is in the Read (high) state when the PIA is selected for a Read operation.

PIA Enable (E) – The enable pulse, E, is the only timing signal that is supplied to the PIA. Timing of all other signals is referenced to the leading and trailing edges of the E pulse. This signal will normally be a derivative of the MC6800 ϕ2 Clock.

PIA Read/Write (R/W) – This signal is generated by the MPU to control the direction of data transfers on the Data Bus. A low state on the PIA Read/Write line enables the input buffers and data is transferred from the MPU to the PIA on the E signal if the device has been selected. A high on the Read/Write line sets up the PIA for a transfer of data to the bus. The PIA output buffers are enabled when the proper address and the enable pulse E are present.

$\overline{\text{Reset}}$ – The active low $\overline{\text{Reset}}$ line is used to reset all register bits in the PIA to a logical zero (low). This line can be used as a power-on reset and as a master reset during system operation.

PIA Chip Select (CS0, CS1 and $\overline{\text{CS2}}$) These three input signals are used to select the PIA. CS0 and CS1 must be high and $\overline{\text{CS2}}$ must be low for selection of the device. Data transfers are then performed under the control of the Enable and Read/Write signals. The chip select lines must be stable for the duration of the E pulse. The device is deselected when any of the chip selects are in the inactive state.

PIA Register Select (RS0 and RS1) – The two register select lines are used to select the various registers inside the PIA. These two lines are used in conjunction with internal Control Registers to select a particular register that is to be written or read.

The register and chip select lines should be stable for the duration of the E pulse while in the read or write cycle.

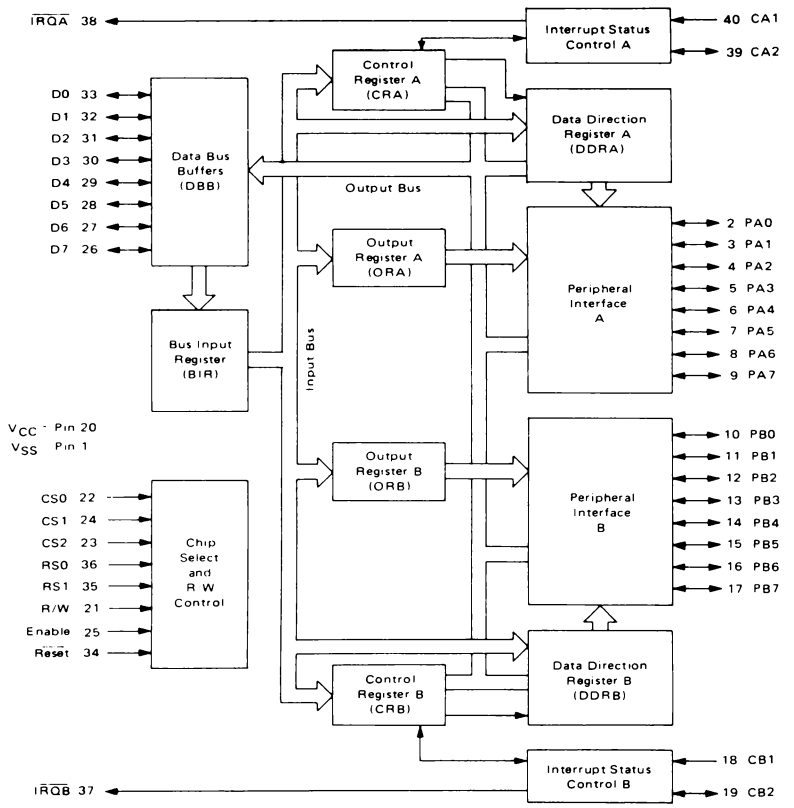
Interrupt Request ($\overline{\text{IRQA}}$ and $\overline{\text{IRQB}}$) The active low Interrupt Request lines ($\overline{\text{IRQA}}$ and $\overline{\text{IRQB}}$) act to interrupt the MPU either directly or through interrupt priority circuitry. These lines are "open drain" (no load device on the chip). This permits all interrupt request lines to be tied together in a wire-OR configuration.

Each Interrupt Request line has two internal interrupt flag bits that can cause the Interrupt Request line to go low. Each flag bit is associated with a particular peripheral interrupt line. Also four interrupt enable bits are provided in the PIA which may be used to inhibit a particular interrupt from a peripheral device.

Servicing an interrupt by the MPU may be accomplished by a software routine that, on a prioritized basis, sequentially reads and tests the two control registers in each PIA for interrupt flag bits that are set.

The interrupt flags are cleared (zeroed) as a result of an

EXPANDED BLOCK DIAGRAM



MPU Read Peripheral Data Operation of the corresponding data register. After being cleared, the interrupt flag bit cannot be enabled to be set until the PIA is deselected during an E pulse. The E pulse is used to condition the interrupt control lines (CA1, CA2, CB1, CB2). When these lines are used as interrupt inputs at least one E pulse must occur from the inactive edge to the active edge of the interrupt input signal to condition the edge sense network. If the interrupt flag has been enabled and the edge sense circuit has been properly conditioned, the interrupt flag will be set on the next active transition of the interrupt input pin.

PIA PERIPHERAL INTERFACE LINES

The PIA provides two 8-bit bi-directional data buses and four interrupt/control lines for interfacing to peripheral devices.

Section A Peripheral Data (PA0-PA7) – Each of the peripheral data lines can be programmed to act as an input or output. This is accomplished by setting a "1" in the corresponding Data Direction Register bit for those lines which are to be outputs. A "0" in a bit of the Data Direction Register causes the corresponding peripheral data line to act as an input. During an MPU Read Peripheral Data Operation, the data on peripheral lines programmed to act as inputs appears directly on the corresponding MPU Data Bus lines. In the input mode the internal pullup resistor on these lines represents a maximum of 1.5 standard TTL loads.

The data in Output Register A will appear on the data lines that are programmed to be outputs. A logical "1" written into the register will cause a "high" on the corresponding data line while a "0" results in a "low". Data in Output Register A may be read by an MPU "Read Peripheral Data A" operation when the corresponding lines are programmed as outputs. This data will be read properly if the voltage on the peripheral data lines is greater than 2.0 volts for a logic "1" output and less than 0.8 volt for a logic "0" output. Loading the output lines such that the voltage on these lines does not reach full voltage causes the data transferred into the MPU on a Read operation to differ from that contained in the respective bit of Output Register A.

Section B Peripheral Data (PB0-PB7) – The peripheral data lines in the B Section of the PIA can be programmed to act as either inputs or outputs in a similar manner to PA0-PA7. However, the output buffers driving these lines differ from those driving lines PA0-PA7. They have three-state capability, allowing them to enter a high impedance state when the peripheral data line is used as an input. In addition, data on the peripheral data lines PB0-PB7 will be read properly from those lines programmed as outputs even if the voltages are below 2.0 volts for a "high". As outputs, these lines are compatible with standard TTL and may also be used as a source of up to 1 milliampere at 1.5 volts to directly drive the base of a transistor switch.

Interrupt Input (CA1 and CB1) – Peripheral Input lines CA1 and CB1 are input only lines that set the interrupt flags of the control registers. The active transition for these signals is also programmed by the two control registers.

Peripheral Control (CA2) – The peripheral control line CA2 can be programmed to act as an interrupt input or as a peripheral control output. As an output, this line is compatible with standard TTL; as an input the internal pullup resistor on this line represents 1.5 standard TTL loads. The function of this signal line is programmed with Control Register A.

Peripheral Control (CB2) – Peripheral Control line CB2 may also be programmed to act as an interrupt input or peripheral control output. As an input, this line has high input impedance and is compatible with standard TTL. As an output it is compatible with standard TTL and may also be used as a source of up to 1 milliampere at 1.5 volts to directly drive the base of a transistor switch. This line is programmed by Control Register B.

INTERNAL CONTROLS

There are six locations within the PIA accessible to the MPU data bus: two Peripheral Registers, two Data Direction Registers, and two Control Registers. Selection of these locations is controlled by the RS0 and RS1 inputs together with bit 2 in the Control Register, as shown in Table 1.

TABLE 1 – INTERNAL ADDRESSING

RS1	RS0	Control Register Bit CRA-2	Control Register Bit CRB-2	Location Selected
0	0	1	X	Peripheral Register A
0	0	0	X	Data Direction Register A
0	1	X	X	Control Register A
1	0	X	1	Peripheral Register B
1	0	X	0	Data Direction Register B
1	1	X	X	Control Register B

X Don't Care

INITIALIZATION

A low reset line has the effect of zeroing all PIA registers. This will set PA0-PA7, PB0-PB7, CA2 and CB2 as inputs, and all interrupts disabled. The PIA must be configured during the restart program which follows the reset.

Details of possible configurations of the Data Direction and Control Register are as follows.

DATA DIRECTION REGISTERS (DDRA and DDRB)

The two Data Direction Registers allow the MPU to control the direction of data through each corresponding peripheral data line. A Data Direction Register bit set at "0" configures the corresponding peripheral data line as an input; a "1" results in an output.

CONTROL REGISTERS (CRA and CRB)

The two Control Registers (CRA and CRB) allow the MPU to control the operation of the four peripheral control lines CA1, CA2, CB1 and CB2. In addition they allow the MPU to enable the interrupt lines and monitor the status of the interrupt flags. Bits 0 through 5 of the two registers may be written or read by the MPU when the proper chip select and register select signals are applied. Bits 6 and 7 of the two registers are read only and are modified by external interrupts occurring on control lines CA1, CA2, CB1 or CB2. The format of the control words is shown in Table 2.

TABLE 2 – CONTROL WORD FORMAT

	7	6	5	4	3	2	1	0
CRA	IRQA1	IRQA2	CA2 Control			DDRA Access	CA1 Control	
	7	6	5	4	3	2	1	0
CRB	IRQB1	IRQB2	CB2 Control			DDRB Access	CB1 Control	

Data Direction Access Control Bit (CRA-2 and CRB-2) – Bit 2 in each Control register (CRA and CRB) allows selection of either a Peripheral Interface Register or the Data Direction Register when the proper register select signals are applied to RS0 and RS1.

Interrupt Flags (CRA-6, CRA-7, CRB-6, and CRB-7) - The four interrupt flag bits are set by active transitions of signals on the four Interrupt and Peripheral Control lines when those lines are programmed to be inputs. These bits cannot be set directly from the MPU Data Bus and are reset indirectly by a Read Peripheral Data Operation on the appropriate section.

TABLE 3 – CONTROL OF INTERRUPT INPUTS CA1 AND CB1

CRA-1 (CRB-1)	CRA-0 (CRB-0)	Interrupt Input CA1 (CB1)	Interrupt Flag CRA-7 (CRB-7)	MPU Interrupt Request $\overline{IRQA}$ ($\overline{IRQB}$)
0	0	↓ Active	Set high on ↓ of CA1 (CB1)	Disabled — $\overline{IRQ}$ remains high
0	1	↓ Active	Set high on ↓ of CA1 (CB1)	Goes low when the interrupt flag bit CRA-7 (CRB-7) goes high
1	0	↑ Active	Set high on ↑ of CA1 (CB1)	Disabled — $\overline{IRQ}$ remains high
1	1	↑ Active	Set high on ↑ of CA1 (CB1)	Goes low when the interrupt flag bit CRA-7 (CRB-7) goes high

Notes 1 ↑ indicates positive transition (low to high)

2 ↓ indicates negative transition (high to low)

3 The Interrupt flag bit CRA-7 is cleared by an MPU Read of the A Data Register, and CRB-7 is cleared by an MPU Read of the B Data Register

4 If CRA-0 (CRB-0) is low when an interrupt occurs (interrupt disabled) and is later brought high, $\overline{IRQA}$ ($\overline{IRQB}$) occurs after CRA-0 (CRB-0) is written to a "one".

Control of CA1 and CB1 Interrupt Input Lines (CRA-0, CRB-0, CRA-1, and CRB-1) – The two lowest order bits of the control registers are used to control the interrupt input lines CA1 and CB1. Bits CRA-0 and CRB-0 are used to enable the MPU interrupt signals $\overline{IRQA}$ and $\overline{IRQB}$, respectively. Bits CRA-1 and CRB-1 determine the active transition of the interrupt input signals CA1 and CB1 (Table 3).

TABLE 4 – CONTROL OF CA2 AND CB2 AS INTERRUPT INPUTS
CRA5 (CRB5) is low

CRA-5 (CRB-5)	CRA-4 (CRB-4)	CRA-3 (CRB-3)	Interrupt Input CA2 (CB2)	Interrupt Flag CRA-6 (CRB-6)	MPU Interrupt Request $\overline{IRQA}$ ($\overline{IRQB}$)
0	0	0	↓ Active	Set high on ↓ of CA2 (CB2)	Disabled — $\overline{IRQ}$ remains high
0	0	1	↓ Active	Set high on ↓ of CA2 (CB2)	Goes low when the interrupt flag bit CRA-6 (CRB-6) goes high
0	1	0	↑ Active	Set high on ↑ of CA2 (CB2)	Disabled — $\overline{IRQ}$ remains high
0	1	1	↑ Active	Set high on ↑ of CA2 (CB2)	Goes low when the interrupt flag bit CRA-6 (CRB-6) goes high

Notes: 1. ↑ indicates positive transition (low to high)

2. ↓ indicates negative transition (high to low)

3. The interrupt flag bit CRA-6 is cleared by an MPU Read of the A Data Register and CRB-6 is cleared by an MPU Read of the B Data Register.

4. If CRA-3 (CRB-3) is low when an interrupt occurs (interrupt disabled) and is later brought high, $\overline{IRQA}$ ($\overline{IRQB}$) occurs after CRA-3 (CRB-3) is written to a "one".

TABLE 5 – CONTROL OF CB2 AS AN OUTPUT
CRB-5 is high

CRB-5	CRB-4	CRB-3	CB2	
			Cleared	Set
1	0	0	Low on the positive transition of the first E pulse following an MPU Write "B" Data Register operation	High when the interrupt flag bit CRB-7 is set by an active transition of the CB1 signal
1	0	1	Low on the positive transition of the first E pulse after an MPU Write "B" Data Register operation.	High on the positive edge of the first "E" pulse following an "E" pulse which occurred while the part was deselected.
1	1	0	Low when CRB-3 goes low as a result of an MPU Write in Control Register "B"	Always low as long as CRB-3 is low. Will go high on an MPU Write in Control Register "B" that changes CRB-3 to "one".
1	1	1	Always high as long as CRB-3 is high. Will be cleared when an MPU Write Control Register "B" results in clearing CRB-3 to "zero".	High when CRB-3 goes high as a result of an MPU Write into Control Register "B".

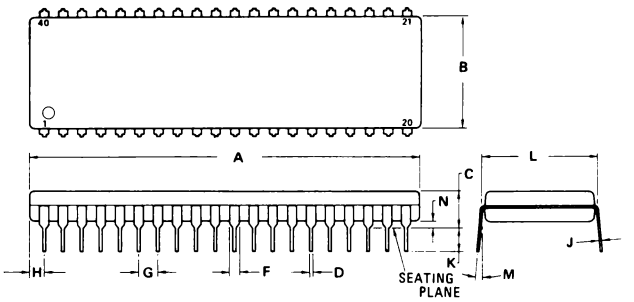
Control of CA2 and CB2 Peripheral Control Lines (CRA-3, CRA-4, CRA-5, CRB-3, CRB-4, and CRB-5) – Bits 3, 4, and 5 of the two control registers are used to control the CA2 and CB2 Peripheral Control lines. These bits determine if the control lines will be an interrupt input or an output control signal. If bit CRA-5 (CRB-5) is low, CA2 (CB2) is an interrupt input line similar to CA1 (CB1) (Table 4). When CRA-5 (CRB-5) is high, CA2 (CB2) becomes an output signal that may be used to control peripheral data transfers. When in the output mode, CA2 and CB2 have slightly different characteristics (Tables 5 and 6).

TABLE 6 – CONTROL OF CA-2 AS AN OUTPUT
CRA-5 is high

CRA-5	CRA-4	CRA-3	CA2	
			Cleared	Set
1	0	0	Low on negative transition of E after an MPU Read "A" Data operation.	High when the interrupt flag bit CRA-7 is set by an active transition of the CA1 signal.
1	0	1	Low on negative transition of E after an MPU Read "A" Data operation.	High on the negative edge of the first "E" pulse which occurs during a deselect.
1	1	0	Low when CRA-3 goes low as a result of an MPU Write to Control Register "A".	Always low as long as CRA-3 is low. Will go high on an MPU Write to Control Register "A" that changes CRA-3 to "one".
1	1	1	Always high as long as CRA-3 is high. Will be cleared on an MPU Write to Control Register "A" that clears CRA-3 to a "zero".	High when CRA-3 goes high as a result of an MPU Write to Control Register "A".

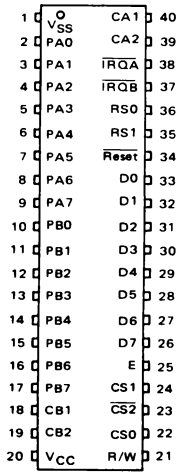
PACKAGE DIMENSIONS

CASE 711-01



DIM	MILLIMETERS MIN	MILLIMETERS MAX	INCHES MIN	INCHES MAX
A	51.82	52.32	2.040	2.060
B	13.72	14.22	0.540	0.560
C	4.57	5.08	0.180	0.200
D	0.36	0.51	0.014	0.020
F	1.02	1.52	0.040	0.060
G	2.41	2.67	0.095	0.105
H	1.65	2.16	0.065	0.085
J	0.20	0.30	0.008	0.012
K	3.68	4.19	0.145	0.165
L	14.99	15.49	0.590	0.610
M	0^0	10^0	0^0	10^0
N	0.51	1.02	0.020	0.040

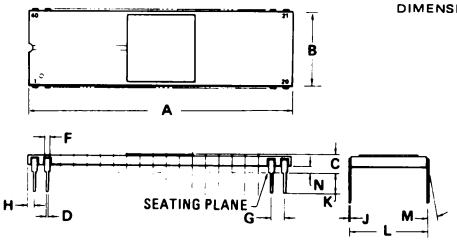
PIN ASSIGNMENT



PACKAGE DIMENSIONS

CASE 715-02
(CERAMIC)

SEE PAGE 165 FOR PLASTIC PACKAGE DIMENSIONS.



DIM	MILLIMETERS MIN	MILLIMETERS MAX	INCHES MIN	INCHES MAX
A	50.29	51.31	1.980	2.020
B	14.86	15.62	0.585	0.615
C	2.54	4.19	0.100	0.165
D	0.38	0.53	0.015	0.021
F	0.76	1.40	0.030	0.055
G	2.54 BSC		0.100 BSC	
H	0.76	1.78	0.030	0.070
J	0.20	0.33	0.008	0.013
K	2.54	4.19	0.100	0.165
L	14.60	15.37	0.575	0.605
M	–	10^0	–	10^0
N	0.51	1.52	0.020	0.060

NOTE:
1. LEADS, TRUE POSITIONED WITHIN 0.25 mm (0.010) DIA (AT SEATING PLANE), AT MAX. MAT'L CONDITION.

MOTOROLA
SEMICONDUCTORS
3501 ED BLUESTEIN BLVD., AUSTIN, TEXAS 78721

MC6850 (1.0 MHz) MC68A50 (1.5 MHz) MC68B50 (2.0 MHz)

MOS
(N-CHANNEL, SILICON-GATE)

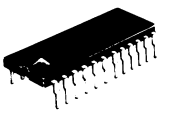
ASYNCHRONOUS COMMUNICATIONS INTERFACE ADAPTER

ASYNCHRONOUS COMMUNICATIONS INTERFACE ADAPTER (ACIA)

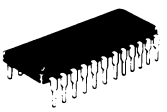
The MC6850 Asynchronous Communications Interface Adapter provides the data formatting and control to interface serial asynchronous data communications information to bus organized systems such as the MC6800 Microprocessing Unit.

The bus interface of the MC6850 includes select, enable, read/write, interrupt and bus interface logic to allow data transfer over an 8-bit bidirectional data bus. The parallel data of the bus system is serially transmitted and received by the asynchronous data interface, with proper formatting and error checking. The functional configuration of the ACIA is programmed via the data bus during system initialization. A programmable Control Register provides variable word lengths, clock division ratios, transmit control, receive control, and interrupt control. For peripheral or modem operation, three control lines are provided. These lines allow the ACIA to interface directly with the MC6860L 0-600 bps digital modem.

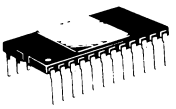
- 8- and 9-Bit Transmission
- Optional Even and Odd Parity
- Parity, Overrun and Framing Error Checking
- Programmable Control Register
- Optional ÷1, ÷16, and ÷64 Clock Modes
- Up to 1.0 Mbps Transmission
- False Start Bit Deletion
- Peripheral/Modem Control Functions
- Double Buffered
- One- or Two-Stop Bit Operation



S SUFFIX
CERDIP PACKAGE
CASE 623

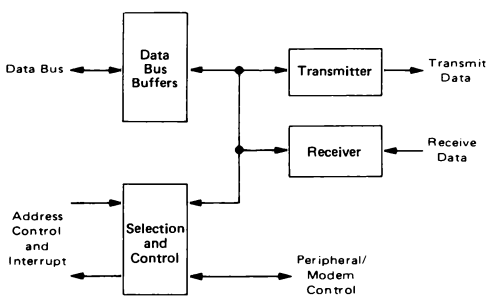


P SUFFIX
PLASTIC PACKAGE
CASE 709

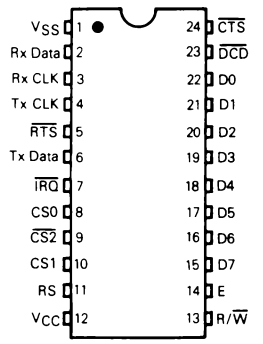


L SUFFIX
CERAMIC PACKAGE
CASE 716

MC6850 ASYNCHRONOUS COMMUNICATIONS INTERFACE ADAPTER BLOCK DIAGRAM



PIN ASSIGNMENT



 DS 9493-R2

MAXIMUM RATINGS

Characteristics	Symbol	Value	Unit
Supply Voltage	V_{CC}	−0.3 to +7.0	V
Input Voltage	V_{in}	−0.3 to +7.0	V
Operating Temperature Range MC6850, MC68A50, MC68B50 MC6850C, MC68A50C, MC68B50C	T_A	T_L to T_H 0 to 70 −40 to +85	°C
Storage Temperature Range	T_{stg}	−55 to +150	°C

This device contains circuitry to protect the inputs against damage due to high static voltages or electric fields; however, it is advised that normal precautions be taken to avoid application of any voltage higher than maximum rated voltages to this high-impedance circuit. Reliability of operation is enhanced if unused inputs are tied to an appropriate logic voltage level (e.g., either V_{SS} or V_{CC}).

THERMAL CHARACTERISTICS

Characteristic	Symbol	Value	Unit
Thermal Resistance Plastic Ceramic Cerdip	θ_{JA}	 120 60 65	°C/W

POWER CONSIDERATIONS

The average chip-junction temperature, T_J, in °C can be obtained from:

$$T_J = T_A + (P_D \bullet \theta_{JA}) \quad (1)$$

Where:

T_A = Ambient Temperature, °C

θ_{JA} = Package Thermal Resistance, Junction-to-Ambient, °C/W

$P_D = P_{INT} + P_{PORT}$

$P_{INT} = I_{CC} \times V_{CC}$, Watts — Chip Internal Power

P_{PORT} = Port Power Dissipation, Watts — User Determined

For most applications $P_{PORT} \ll P_{INT}$ and can be neglected. P_{PORT} may become significant if the device is configured to drive Darlington bases or sink LED loads.

An approximate relationship between P_D and T_J (if P_{PORT} is neglected) is:

$$P_D = K \div (T_J + 273°C) \quad (2)$$

Solving equations 1 and 2 for K gives:

$$K = P_D \bullet (T_A + 273°C) + \theta_{JA} \bullet P_D^2 \quad (3)$$

Where K is a constant pertaining to the particular part. K can be determined from equation 3 by measuring P_D (at equilibrium) for a known T_A. Using this value of K the values of P_D and T_J can be obtained by solving equations (1) and (2) iteratively for any value of T_A.

DC ELECTRICAL CHARACTERISTICS (V_{CC} = 5.0 Vdc ±5%, V_{SS} = 0, $T_A = T_L$ to T_H unless otherwise noted.)

Characteristic		Symbol	Min	Typ	Max	Unit
Input High Voltage		V_{IH}	V_{SS} + 2.0	–	V_{CC}	V
Input Low Voltage		V_{IL}	V_{SS} − 0.3	–	V_{SS} + 0.8	V
Input Leakage Current (V_{in} = 0 to 5.25 V)	R/$\overline{W}$, CS0, CS1, $\overline{CS2}$, Enable RS, Rx D, Rx C, $\overline{CTS}$, $\overline{DCD}$	I_{in}	–	1.0	2.5	μA
Three-State (Off State) Input Current (V_{in} = 0.4 to 2.4 V)	D0-D7	I_{TSI}	–	2.0	10	μA
Output High Voltage (I_{Load} = −205 μA, Enable Pulse Width < 25 μs) (I_{Load} = −100 μA, Enable Pulse Width < 25 μs)	D0-D7 Tx Data, $\overline{RTS}$	V_{OH}	V_{SS} + 2.4 V_{SS} + 2.4	– –	– –	V
Output Low Voltage (I_{Load} = 1.6 mA, Enable Pulse Width < 25 μs)		V_{OL}	–	–	V_{SS} + 0.4	V
Output Leakage Current (Off State) (V_{OH} = 2.4 V)	$\overline{IRQ}$	I_{LOH}	–	1.0	10	μA
Internal Power Dissipation (Measured at $T_A = T_L$)		P_{INT}	–	300	525	mW
Internal Input Capacitance (V_{in} = 0, T_A = 25°C, f = 1.0 MHz)	D0-D7 E, Tx CLK, Rx CLK, R/W, RS, Rx Data, CS0, CS1, $\overline{CS2}$, $\overline{CTS}$, $\overline{DCD}$	C_{in}	– –	10 7.0	12.5 7.5	pF
Output Capacitance (V_{in} = 0, T_A = 25°C, f = 1.0 MHz)	RTS, Tx Data $\overline{IRQ}$	C_{out}	– –	– –	10 5.0	pF

SERIAL DATA TIMING CHARACTERISTICS

Characteristic		Symbol	MC6850		MC68A50		MC68B50		Unit
			Min	Max	Min	Max	Min	Max	
Data Clock Pulse Width, Low (See Figure 1)	÷16, ÷64 Modes ÷1 Mode	PW_{CL}	600 900	– –	450 650	– –	280 500	– –	ns
Data Clock Pulse Width, High (See Figure 2)	÷16, ÷64 Modes ÷1 Mode	PW_{CH}	600 900	– –	450 650	– –	280 500	– –	ns
Data Clock Frequency	÷16, ÷64 Modes ÷1 Mode	f_C	– –	0.8 500	– –	1.0 750	– –	1.5 1000	MHz kHz
Data Clock-to-Data Delay for Transmitter (See Figure 3)		t_{TDD}	–	600	–	540	–	460	ns
Receive Data Setup Time (See Figure 4)	÷1 Mode	t_{RDS}	250	–	100	–	30	–	ns
Receive Data Hold Time (See Figure 5)	÷1 Mode	t_{RDH}	250	–	100	–	30	–	ns
Interrupt Request Release Time (See Figure 6)		t_{IR}	–	1.2	–	0.9	–	0.7	μs
Request-to-Send Delay Time (See Figure 6)		t_{RTS}	–	560	–	480	–	400	ns
Input Rise and Fall Times (or 10% of the pulse width if smaller)		t_r, t_f	–	1.0	–	0.5	–	0.25	μs

FIGURE 1 – CLOCK PULSE WIDTH, LOW-STATE

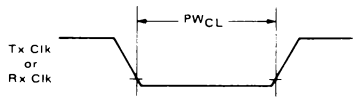


FIGURE 2 – CLOCK PULSE WIDTH, HIGH-STATE

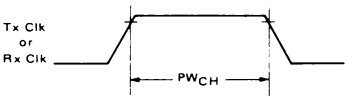


FIGURE 3 – TRANSMIT DATA OUTPUT DELAY

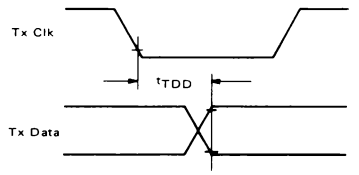


FIGURE 4 – RECEIVE DATA SETUP TIME (÷1 Mode)

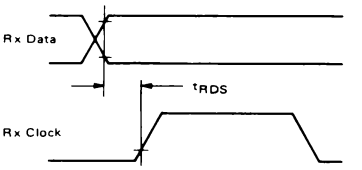


FIGURE 5 – RECEIVE DATA HOLD TIME (÷1 Mode)

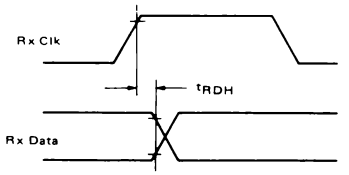
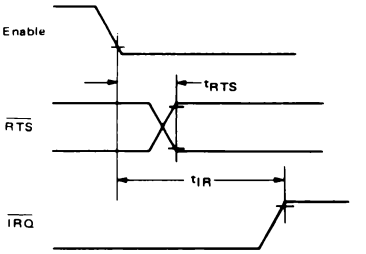


FIGURE 6 – REQUEST-TO-SEND DELAY AND INTERRUPT-REQUEST RELEASE TIMES



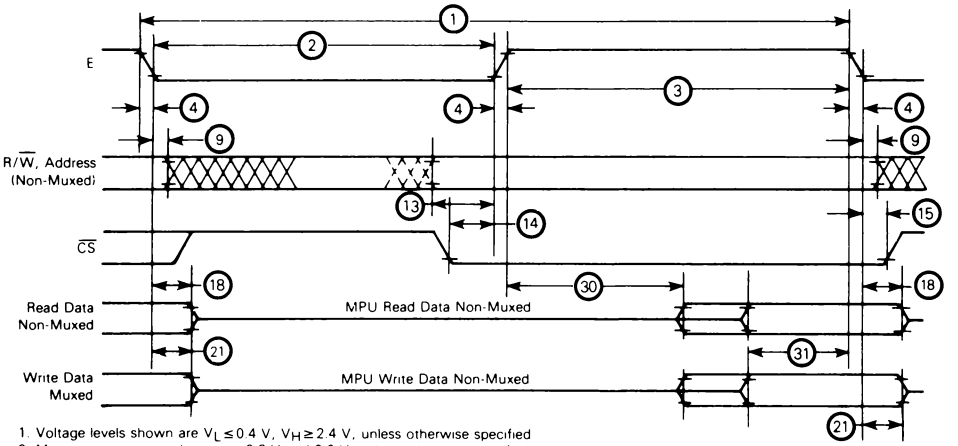
Note: Timing measurements are referenced to and from a low voltage of 0.8 volts and a high voltage of 2.0 volts, unless otherwise noted.

BUS TIMING CHARACTERISTICS (See Notes 1 and 2 and Figure 7)

Ident. Number	Characteristic	Symbol	MC6850		MC68A50		MC68B50		Unit
			Min	Max	Min	Max	Min	Max	
1	Cycle Time	t_{cyc}	1.0	10	0.67	10	0.5	10	μs
2	Pulse Width, E Low	PW_{EL}	430	9500	280	9500	210	9500	ns
3	Pulse Width, E High	PW_{EH}	450	9500	280	9500	220	9500	ns
4	Clock Rise and Fall Time	t_r, t_f	–	25	–	25	–	20	ns
9	Address Hold Time	t_{AH}	10	–	10	–	10	–	ns
13	Address Setup Time Before E	t_{AS}	80	–	60	–	40	–	ns
14	Chip Select Setup Time Before E	t_{CS}	80	–	60	–	40	–	ns
15	Chip Select Hold Time	t_{CH}	10	–	10	–	10	–	ns
18	Read Data Hold Time	t_{DHR}	20	50*	20	50*	20	50*	ns
21	Write Data Hold Time	t_{DHW}	10	–	10	–	10	–	ns
30	Output Data Delay Time	t_{DDR}	–	290	–	180	–	150	ns
31	Input Data Setup Time	t_{DSW}	165	–	80	–	60	–	ns

*The data bus output buffers are no longer sourcing or sinking current by t_{DHR}max (High Impedance).

FIGURE 7 — BUS TIMING CHARACTERISTICS



1. Voltage levels shown are $V_L \leq 0.4$ V, $V_H \geq 2.4$ V, unless otherwise specified
2. Measurement points shown are 0.8 V and 2.0 V, unless otherwise specified

FIGURE 8 — BUS TIMING TEST LOADS

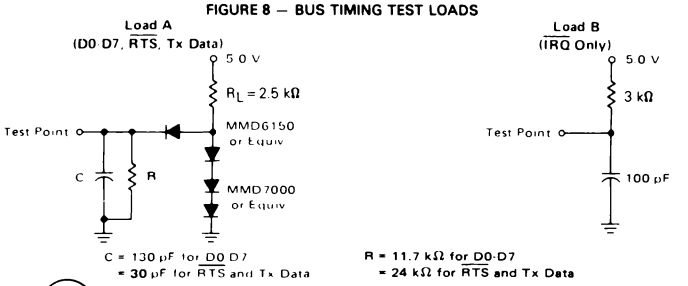
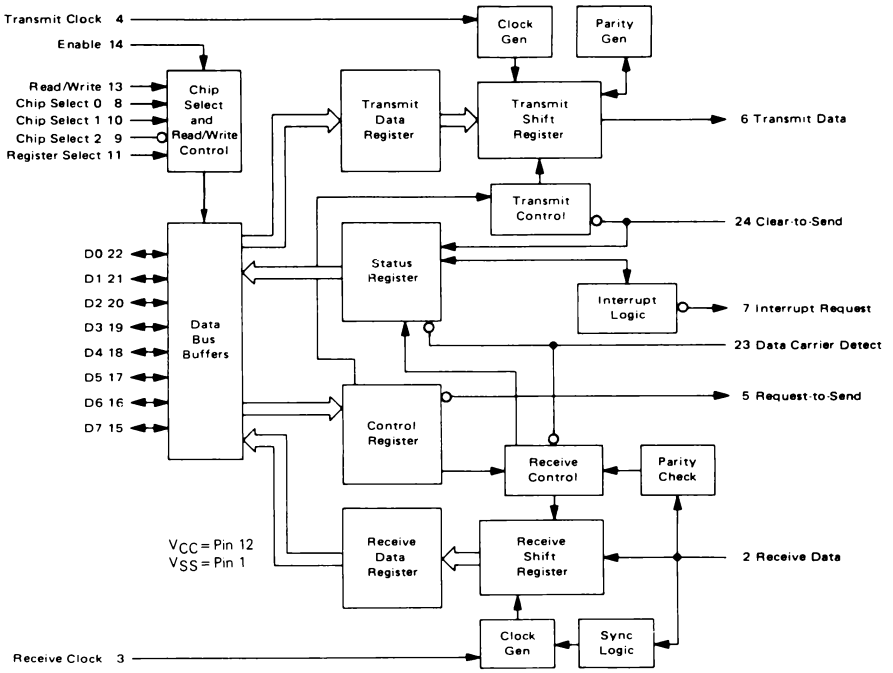


FIGURE 9 — EXPANDED BLOCK DIAGRAM



DEVICE OPERATION

At the bus interface, the ACIA appears as two addressable memory locations. Internally, there are four registers: two read-only and two write-only registers. The read-only registers are Status and Receive Data; the write-only registers are Control and Transmit Data. The serial interface consists of serial input and output lines with independent clocks, and three peripheral/modem control lines.

POWER ON/MASTER RESET

The master reset (CR0, CR1) should be set during system initialization to insure the reset condition and prepare for programming the ACIA functional configuration when the communications channel is required. During the first master reset, the $\overline{IRQ}$ and $\overline{RTS}$ outputs are held at level 1. On all other master resets, the $\overline{RTS}$ output can be programmed high or low with the $\overline{IRQ}$ output held high. Control bits CR5 and CR6 should also be programmed to define the state of $\overline{RTS}$ whenever master reset is utilized. The ACIA also contains internal power-on reset logic to detect the power line turn-on transition and hold the chip in a reset state to prevent erroneous output transitions prior to initialization. This circuitry depends on clean power turn-on transitions. The power-on reset is released by means of the bus-programmed master reset which must be applied prior to operating the ACIA. After master resetting the ACIA, the programmable Control Register can be set for a number of options such as variable clock divider ratios, variable word length, one or two stop bits, parity (even, odd, or none), etc.

TRANSMIT

A typical transmitting sequence consists of reading the ACIA Status Register either as a result of an interrupt or in the ACIA's turn in a polling sequence. A character may be written into the Transmit Data Register if the status read operation has indicated that the Transmit Data Register is empty. This character is transferred to a Shift Register where it is serialized and transmitted from the Transmit Data output preceded by a start bit and followed by one or two stop bits. Internal parity (odd or even) can be optionally added to the character and will occur between the last data bit and the first stop bit. After the first character is written in the Data Register, the Status Register can be read again to check for a Transmit Data Register Empty condition and current peripheral status. If the register is empty, another character can be loaded for transmission even though the first character is in the process of being transmitted (because of

double buffering). The second character will be automatically transferred into the Shift Register when the first character transmission is completed. This sequence continues until all the characters have been transmitted.

RECEIVE

Data is received from a peripheral by means of the Receive Data input. A divide-by-one clock ratio is provided for an externally synchronized clock (to its data) while the divide-by-16 and 64 ratios are provided for internal synchronization. Bit synchronization in the divide-by-16 and 64 modes is initiated by the detection of 8 or 32 low samples on the receive line in the divide-by-16 and 64 modes respectively. False start bit deletion capability insures that a full half bit of a start bit has been received before the internal clock is synchronized to the bit time. As a character is being received, parity (odd or even) will be checked and the error indication will be available in the Status Register along with framing error, overrun error, and Receive Data Register full. In a typical receiving sequence, the Status Register is read to determine if a character has been received from a peripheral. If the Receiver Data Register is full, the character is placed on the 8-bit ACIA bus when a Read Data command is received from the MPU. When parity has been selected for a 7-bit word (7 bits plus parity), the receiver strips the parity bit (D7 = 0) so that data alone is transferred to the MPU. This feature reduces MPU programming. The Status Register can continue to be read to determine when another character is available in the Receive Data Register. The receiver is also double buffered so that a character can be read from the data register as another character is being received in the shift register. The above sequence continues until all characters have been received.

INPUT/OUTPUT FUNCTIONS

ACIA INTERFACE SIGNALS FOR MPU

The ACIA interfaces to the M6800 MPU with an 8-bit bidirectional data bus, three chip select lines, a register select line, an interrupt request line, read/write line, and enable line. These signals permit the MPU to have complete control over the ACIA.

ACIA Bidirectional Data (D0-D7) – The bidirectional data lines (D0-D7) allow for data transfer between the ACIA and the MPU. The data bus output drivers are three-state devices that remain in the high-impedance (off) state except when the MPU performs an ACIA read operation.

ACIA Enable (E) – The Enable signal, E, is a high-impedance TTL-compatible input that enables the bus input/output data buffers and clocks data to and from the ACIA. This signal will normally be a derivative of the MC6800 $\phi 2$ Clock or MC6809 E clock.

Read/Write (R/$\overline{W}$) – The Read/Write line is a high-impedance input that is TTL compatible and is used to control the direction of data flow through the ACIA's input/output data bus interface. When Read/Write is high (MPU Read cycle), ACIA output drivers are turned on and a selected register is read. When it is low, the ACIA output drivers are turned off and the MPU writes into a selected register. Therefore, the Read/Write signal is used to select read-only or write-only registers within the ACIA.

Chip Select (CS0, CS1, $\overline{CS2}$) – These three high-impedance TTL-compatible input lines are used to address the ACIA. The ACIA is selected when CS0 and CS1 are high and $\overline{CS2}$ is low. Transfers of data to and from the ACIA are then performed under the control of the Enable Signal, Read/Write, and Register Select.

Register Select (RS) – The Register Select line is a high-impedance input that is TTL compatible. A high level is used to select the Transmit/Receive Data Registers and a low level the Control/Status Registers. The Read/Write signal line is used in conjunction with Register Select to select the read-only or write-only register in each register pair.

Interrupt Request ($\overline{IRQ}$) – Interrupt Request is a TTL-compatible, open-drain (no internal pullup), active low output that is used to interrupt the MPU. The $\overline{IRQ}$ output remains low as long as the cause of the interrupt is present and the appropriate interrupt enable within the ACIA is set. The $\overline{IRQ}$ status bit, when high, indicates the $\overline{IRQ}$ output is in the active state.

Interrupts result from conditions in both the transmitter and receiver sections of the ACIA. The transmitter section causes an interrupt when the Transmitter Interrupt Enabled condition is selected (CR5•$\overline{CR6}$), and the Transmit Data Register Empty (TDRE) status bit is high. The TDRE status bit indicates the current status of the Transmitter Data Register except when inhibited by Clear-to-Send ($\overline{CTS}$) being high or the ACIA being maintained in the Reset condition. The interrupt is cleared by writing data into the Transmit Data Register. The interrupt is masked by disabling the Transmitter Interrupt via CR5 or CR6 or by the loss of $\overline{CTS}$ which inhibits the TDRE status bit. The Receiver section causes an interrupt when the Receiver Interrupt Enable is set and the Receive Data Register Full (RDRF) status bit is high, an Overrun has occurred, or Data Carrier Detect ($\overline{DCD}$) has gone high. An interrupt resulting from the RDRF status bit can be cleared by reading data or resetting the ACIA. Interrupts caused by Overrun or loss of $\overline{DCD}$ are cleared by reading the status register after the error condition has occurred and then reading the Receive Data Register or resetting the ACIA. The receiver interrupt is masked by resetting the Receiver Interrupt Enable.

CLOCK INPUTS

Separate high-impedance TTL-compatible inputs are provided for clocking of transmitted and received data. Clock frequencies of 1, 16, or 64 times the data rate may be selected.

Transmit Clock (Tx CLK) – The Transmit Clock input is used for the clocking of transmitted data. The transmitter initiates data on the negative transition of the clock.

Receive Clock (Rx CLK) – The Receive Clock input is used for synchronization of received data. (In the ÷1 mode, the clock and data must be synchronized externally.) The receiver samples the data on the positive transition of the clock.

SERIAL INPUT/OUTPUT LINES

Receive Data (Rx Data) – The Receive Data line is a high-impedance TTL-compatible input through which data is received in a serial format. Synchronization with a clock for detection of data is accomplished internally when clock rates of 16 or 64 times the bit rate are used.

Transmit Data (Tx Data) – The Transmit Data output line transfers serial data to a modem or other peripheral.

PERIPHERAL/MODEM CONTROL

The ACIA includes several functions that permit limited control of a peripheral or modem. The functions included are Clear-to-Send, Request-to-Send and Data Carrier Detect.

Clear-to-Send ($\overline{CTS}$) – This high-impedance TTL-compatible input provides automatic control of the transmitting end of a communications link via the modem Clear-to-Send active low output by inhibiting the Transmit Data Register Empty (TDRE) status bit.

Request-to-Send ($\overline{RTS}$) – The Request-to-Send output enables the MPU to control a peripheral or modem via the data bus. The RTS output corresponds to the state of the Control Register bits CR5 and CR6. When CR6 = 0 or both CR5 and CR6 = 1, the $\overline{RTS}$ output is low (the active state). This output can also be used for Data Terminal Ready (DTR).

Data Carrier Detect ($\overline{DCD}$) – This high-impedance TTL-compatible input provides automatic control, such as in the receiving end of a communications link by means of a modem Data Carrier Detect output. The $\overline{DCD}$ input inhibits and initializes the receiver section of the ACIA when high. A low-to-high transition of the Data Carrier Detect initiates an interrupt to the MPU to indicate the occurrence of a loss of carrier when the Receive Interrupt Enable bit is set. The Rx CLK must be running for proper $\overline{DCD}$ operation.

ACIA REGISTERS

The expanded block diagram for the ACIA indicates the internal registers on the chip that are used for the status, control, receiving, and transmitting of data. The content of each of the registers is summarized in Table 1.

TRANSMIT DATA REGISTER (TDR)

Data is written in the Transmit Data Register during the negative transition of the enable (E) when the ACIA has been addressed with RS high and R/$\overline{W}$ low. Writing data into the register causes the Transmit Data Register Empty bit in the Status Register to go low. Data can then be transmitted. If the transmitter is idling and no character is being transmitted, then the transfer will take place within 1-bit time of the trailing edge of the Write command. If a character is being transmitted, the new data character will commence as soon as the previous character is complete. The transfer of data causes the Transmit Data Register Empty (TDRE) bit to indicate empty.

RECEIVE DATA REGISTER (RDR)

Data is automatically transferred to the empty Receive Data Register (RDR) from the receiver deserializer (a shift register) upon receiving a complete character. This event causes the Receive Data Register Full bit (RDRF) in the status buffer to go high (full). Data may then be read through the bus by addressing the ACIA and selecting the Receive Data Register with RS and R/$\overline{W}$ high when the ACIA is enabled. The non-destructive read cycle causes the RDRF bit to be cleared to empty although the data is retained in the RDR. The status is maintained by RDRF as to whether or not the data is current. When the Receive Data Register is full, the automatic transfer of data from the Receiver Shift Register to the Data Register is inhibited and the RDR contents remain valid with its current status stored in the Status Register.

TABLE 1 – DEFINITION OF ACIA REGISTER CONTENTS

Data Bus Line Number	Buffer Address			
	RS • R/$\overline{W}$ Transmit Data Register (Write Only)	RS • R/$\overline{W}$ Receive Data Register (Read Only)	$\overline{RS}$ • R/$\overline{W}$ Control Register (Write Only)	$\overline{RS}$ • R/$\overline{W}$ Status Register (Read Only)
0	Data Bit 0*	Data Bit 0	Counter Divide Select 1 (CR0)	Receive Data Register Full (RDRF)
1	Data Bit 1	Data Bit 1	Counter Divide Select 2 (CR1)	Transmit Data Register Empty (TDRE)
2	Data Bit 2	Data Bit 2	Word Select 1 (CR2)	Data Carrier Detect ($\overline{DCD}$)
3	Data Bit 3	Data Bit 3	Word Select 2 (CR3)	Clear to Send ($\overline{CTS}$)
4	Data Bit 4	Data Bit 4	Word Select 3 (CR4)	Framing Error (FE)
5	Data Bit 5	Data Bit 5	Transmit Control 1 (CR5)	Receiver Overrun (OVRN)
6	Data Bit 6	Data Bit 6	Transmit Control 2 (CR6)	Parity Error (PE)
7	Data Bit 7***	Data Bit 7**	Receive Interrupt Enable (CR7)	Interrupt Request ($\overline{IRQ}$)

* Leading bit – LSB – Bit 0
** Data bit will be zero in 7 bit plus parity modes
*** Data bit is "don't care" in 7 bit plus parity modes

CONTROL REGISTER

The ACIA Control Register consists of eight bits of write-only buffer that are selected when RS and R/$\overline{W}$ are low. This register controls the function of the receiver, transmitter, interrupt enables, and the Request-to-Send peripheral/modem control output.

Counter Divide Select Bits (CR0 and CR1) – The Counter Divide Select Bits (CR0 and CR1) determine the divide ratios utilized in both the transmitter and receiver sections of the ACIA. Additionally, these bits are used to provide a master reset for the ACIA which clears the Status Register (except for external conditions on $\overline{CTS}$ and $\overline{DCD}$) and initializes both the receiver and transmitter. Master reset does not affect other Control Register bits. Note that after power-on or a power fail/restart, these bits must be set high to reset the ACIA. After resetting, the clock divide ratio may be selected. These counter select bits provide for the following clock divide ratios:

CR1	CR0	Function
0	0	÷ 1
0	1	÷ 16
1	0	÷ 64
1	1	Master Reset

Word Select Bits (CR2, CR3, and CR4) – The Word Select bits are used to select word length, parity, and the number of stop bits. The encoding format is as follows:

CR4	CR3	CR2	Function
0	0	0	7 Bits + Even Parity + 2 Stop Bits
0	0	1	7 Bits + Odd Parity + 2 Stop Bits
0	1	0	7 Bits + Even Parity + 1 Stop Bit
0	1	1	7 Bits + Odd Parity + 1 Stop Bit
1	0	0	8 Bits + 2 Stop Bits
1	0	1	8 Bits + 1 Stop Bit
1	1	0	8 Bits + Even parity + 1 Stop Bit
1	1	1	8 Bits + Odd Parity + 1 Stop Bit

Word length, Parity Select, and Stop Bit changes are not buffered and therefore become effective immediately.

Transmitter Control Bits (CR5 and CR6) – Two Transmitter Control bits provide for the control of the interrupt from the Transmit Data Register Empty condition, the Request-to-Send ($\overline{RTS}$) output, and the transmission of a Break level (space). The following encoding format is used:

CR6	CR5	Function
0	0	$\overline{RTS}$ = low, Transmitting Interrupt Disabled.
0	1	$\overline{RTS}$ = low, Transmitting Interrupt Enabled.
1	0	$\overline{RTS}$ = high, Transmitting Interrupt Disabled.
1	1	$\overline{RTS}$ = low, Transmits a Break level on the Transmit Data Output. Transmitting Interrupt Disabled.

Receive Interrupt Enable Bit (CR7) – The following interrupts will be enabled by a high level in bit position 7 of the Control Register (CR7): Receive Data Register Full, Overrun, or a low-to-high transition on the Data Carrier Detect ($\overline{DCD}$) signal line.

STATUS REGISTER

Information on the status of the ACIA is available to the MPU by reading the ACIA Status Register. This read-only register is selected when RS is low and R/$\overline{W}$ is high. Information stored in this register indicates the status of the Transmit Data Register, the Receive Data Register and error logic, and the peripheral/modem status inputs of the ACIA.

Receive Data Register Full (RDRF), Bit 0 – Receive Data Register Full indicates that received data has been transferred to the Receive Data Register. RDRF is cleared after an MPU read of the Receive Data Register or by a master reset. The cleared or empty state indicates that the contents of the Receive Data Register are not current. Data Carrier Detect being high also causes RDRF to indicate empty.

Transmit Data Register Empty (TDRE), Bit 1 – The Transmit Data Register Empty bit being set high indicates that the Transmit Data Register contents have been transferred and that new data may be entered. The low state indicates that the register is full and that transmission of a new character has not begun since the last write data command.

Data Carrier Detect ($\overline{DCD}$), Bit 2 – The Data Carrier Detect bit will be high when the $\overline{DCD}$ input from a modem has gone high to indicate that a carrier is not present. This bit going high causes an Interrupt Request to be generated when the Receive Interrupt Enable is set. It remains high after the $\overline{DCD}$ input is returned low until cleared by first reading the Status Register and then the Data Register or until a master reset occurs. If the $\overline{DCD}$ input remains high after read status and read data or master reset has occurred, the interrupt is cleared, the $\overline{DCD}$ status bit remains high and will follow the $\overline{DCD}$ input.

Clear-to-Send ($\overline{CTS}$), Bit 3 – The Clear-to-Send bit indicates the state of the Clear-to-Send input from a modem. A low $\overline{CTS}$ indicates that there is a Clear-to-Send from the modem. In the high state, the Transmit Data Register Empty bit is inhibited and the Clear-to-Send status bit will be high. Master reset does not affect the Clear-to-Send status bit.

Framing Error (FE), Bit 4 – Framing error indicates that the received character is improperly framed by a start and a stop bit and is detected by the absence of the first stop bit. This error indicates a synchronization error, faulty transmission, or a break condition. The framing error flag is set or reset during the receive data transfer time. Therefore, this error indicator is present throughout the time that the associated character is available.

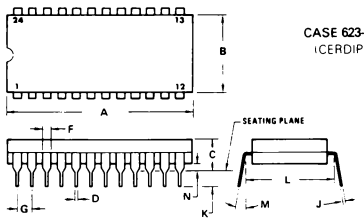
Receiver Overrun (OVRN), Bit 5 – Overrun is an error flag that indicates that one or more characters in the data stream were lost. That is, a character or a number of characters were received but not read from the Receive Data Register (RDR) prior to subsequent characters being received. The overrun condition begins at the midpoint of the last bit of the second character received in succession without a read of the RDR having occurred. The Overrun does not occur in the Status Register until the valid character prior to Overrun has

been read. The RDRF bit remains set until the Overrun is reset. Character synchronization is maintained during the Overrun condition. The Overrun indication is reset after the reading of data from the Receive Data Register or by a Master Reset.

Parity Error (PE), Bit 6 – The parity error flag indicates that the number of highs (ones) in the character does not agree with the preselected odd or even parity. Odd parity is defined to be when the total number of ones is odd. The parity error indication will be present as long as the data character is in the RDR. If no parity is selected, then both the transmitter parity generator output and the receiver partiy check results are inhibited.

Interrupt Request ($\overline{IRQ}$), Bit 7 – The $\overline{IRQ}$ bit indicates the state of the $\overline{IRQ}$ output. Any interrupt condition with its applicable enable will be indicated in this status bit. Anytime the $\overline{IRQ}$ output is low the $\overline{IRQ}$ bit will be high to indicate the interrupt or service request status. $\overline{IRQ}$ is cleared by a read operation to the Receive Data Register or a write operation to the Transmit Data Register.

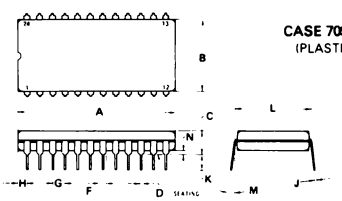
PACKAGE DIMENSIONS



CASE 623-03
(CERDIP)

DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	31.24	32.77	1.230	1.290
B	12.70	15.49	0.500	0.610
C	4.06	5.59	0.160	0.220
D	0.41	0.51	0.016	0.020
F	1.27	1.52	0.050	0.060
G	2.54 BSC		0.100 BSC	
J	0.20	0.30	0.008	0.012
K	2.29	4.06	0.090	0.160
L	15.24 BSC		0.600 BSC	
M	0°	15°	0°	15°
N	0.51	1.27	0.020	0.050

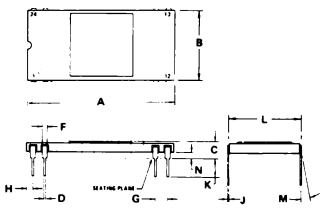
NOTES:
1. DIM "L" TO CENTER OF LEADS WHEN FORMED PARALLEL.
2. LEADS WITHIN 0.13 mm (0.005) RADIUS OF TRUE POSITION AT SEATING PLANE AT MAXIMUM MATERIAL CONDITION. (WHEN FORMED PARALLEL)



CASE 709-02
(PLASTIC)

DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	31.37	32.13	1.235	1.265
B	13.72	14.22	0.540	0.560
C	3.94	5.08	0.155	0.200
D	0.36	0.56	0.014	0.022
F	1.02	1.52	0.040	0.060
G	2.54 BSC		0.100 BSC	
H	1.65	2.03	0.065	0.080
J	0.20	0.38	0.008	0.015
K	2.92	3.43	0.115	0.135
L	15.24 BSC		0.600 BSC	
M	0°	15°	0°	15°
N	0.51	1.02	0.020	0.040

NOTES:
1. POSITIONAL TOLERANCE OF LEADS (D), SHALL BE WITHIN 0.25 mm (0.010) AT MAXIMUM MATERIAL CONDITION, IN RELATION TO SEATING PLANE AND EACH OTHER.
2. DIMENSION L TO CENTER OF LEADS WHEN FORMED PARALLEL.
3. DIMENSION B DOES NOT INCLUDE MOLD FLASH.

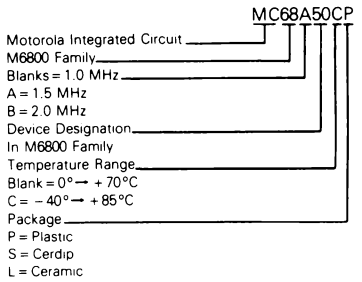


CASE 716-06
(CERAMIC)

DIM	MILLIMETERS		INCHES	
	MIN	MAX	MIN	MAX
A	27.64	30.99	1.088	1.220
B	14.73	15.34	0.580	0.604
C	2.67	4.32	0.105	0.170
D	0.38	0.53	0.015	0.021
F	0.76	1.40	0.030	0.055
G	2.54 BSC		0.100 BSC	
H	0.76	1.78	0.030	0.070
J	0.20	0.30	0.008	0.012
K	2.54	4.57	0.100	0.180
L	14.99	15.49	0.590	0.610
M	–	10°	–	10°
N	1.02	1.52	0.040	0.060

NOTE:
1. LEADS TRUE POSITIONED WITHIN 0.25mm (0.010) DIA (AT SEATING PLANE) AT MAXIMUM MATERIAL CONDITION.
2. DIM "L" TO CENTER OF LEADS WHEN FORMED PARALLEL.

ORDERING INFORMATION



BETTER PROGRAM

Better program processing is available on all types listed. Add suffix letters to part number.

Level 1 add "S" Level 2 add "D" Level 3 add "DS"

Level 1 "S" = 10 Temp Cycles − (−25 to 150°C); Hi Temp testing at T_A max
Level 2 "D" = 168 Hour Burn-in at 125°C
Level 3 "DS" = Combination of Level 1 and 2

Speed	Device	Temperature Range
1.0 MHz	MC6850P,L,S MC6850CP,CL,CS	0 to 70°C −40 to +85°C
1.5 MHz	MC68A50P,L,S MC68A50CP,CL,CS	0 to +70°C −40 to +85°C
2.0 MHz	MC68B50P,L,S	0 to +70°C

MOTOROLA *Semiconductor Products Inc.*

3501 ED BLUESTEIN BLVD., AUSTIN, TEXAS 78721 • A SUBSIDIARY OF MOTOROLA INC.

Index

Absolute addressing, 72, 78–79
  indexed, (See Indexed addressing)
Accumulator A, 11, 13, 27–29
Accumulator addressing, 75
ACIA, 182–93
  control register, 184–88
  receive data register, 184–85, 188–89
  set up, 187–88
  status register, 184–85, 188–91
  transmit data register, 184–85, 189–90
A/D converter, 137–40
Addition, 28, 31
Address bus, 17
Addressing modes, 73–85
Alternate text and graphics pages, 136–39
Analog game controller input (See game controller input)
AND, 76
Annunciator outputs, 134–35, 137
Apple II, 3–5
APPLE BASIC (See BASIC)
Arithmetic
  binary, 39–48
  decimal, 32, 48–50
  6502, 39–50
ASCHEX, 116–17
ASCII
  codes, 24–25, 71–72, 88–89, 92–93, 112
  data, 17, 22–26
  to hex conversion, 116–17
Assembler, 29, 74
Assembler directives, 66, 74, 96
Assembly language, 1–2, 29, 74
Asynchronous communications interface adapter (See ACIA)

Backspace key, 14
BASCLC, 91
BASIC, 205
Band rate, 183
BCD, 32, 48
  addition, 49
  subtraction, 50
Binary
  addition, 39–41
  number, 6–10
  subtraction, 46–48
Binary coded decimal, (See BCD)
Bit time, 183
Bits, 6
BKGND, 124–25, 133
BLOAD, 200
Branching instructions, 51–56

Branching offsets, 53–54, 199
Break flag, 31, 33
BREAK instruction, 173, 176–77
BREAK level, 187
Breakpoints, 67–68, 199
BSAVE, 200
Buffer
  RS232, 191–92
  three-state, 18
Built-in I/O, 149, 151–54
Built-in routines, 102–105
Byte, 5

CALL, 205
Carry, 40, 44–50, 55
  flag, 30–31
CATALOG, 17
Change
  memory, 22
  registers, 13
Character generator, 88–89
Chip select, 18, 145, 147
Clearing the screen, 102–103, 110, 123–24
CLRSCR, 110, 116
CLRTOP, 110, 116
Clock (See real-time clock)
Clock frequency, 62, 186, 191
Command line, 12
Condition code register, 14, 30–33
Condition flags, 30
Conversion:
  hexadecimal to decimal, 9–11
  decimal to hexadecimal, 15–16
Copyright message, 201
COUT, 153, 170
COUT1, 103, 153, 170, 193
CSWH, 153, 170
CSWL, 153, 170
CTRL S, 28, 30

D flag, (See Decimal mode flag)
Data bus, 3, 5
Decimal arithmetic, (See Arithmetic)
Decimal mode flag, 31–32, 39
Decrementing instructions, 36
Delay loops, 62–65
DELETE, 100, 199
DEVICE SELECT, 141–42, 145, 148–49, 151–52
Direct addressing mode, (See zero page addressing mode)
Disassembled code, 94, 172, 199
Disk
  storing data, 200
DOS commands, 201
DRAW, 127–33

Entry line, 13
ESC key, 14, 21
Exclusive-or, 46, 76
Executing programs, 68, 199
Executing single instructions, 28
Expansion ROM, 157–59
Extended addressing mode, (See absolute addressing mode)

FIND, 199
Framing error, 188–89
Frequency
  clock (See Clock frequency)
  of sound wave, 66, 72
Full-duplex, 191

Game controller inputs, 135
Game I/O connector, 134–40
Graphics,
  (See Low-resolution graphics; High-resolution graphics)

Half-duplex, 191
Handshake mode (See PIA)
HCLR, 123
HCOLOR, 125–26
HCOLOR1, 124–25
Hex data, 17, 22
Hexadecimal, 6
  digit, 7
  numbers, 6–10, 14
HGR, 123, 133
HGR2, 123, 133
High-resolution graphics, 118–33
  colors, 124–25
HLIN, 114, 116, 126 –27, 133
HOME, 102
Horizontal bar graphs (*See* Bar graphs)
HPLOT, 125–26, 133
HPOSN, 119, 122–12, 130, 133

Immediate addressing, 75
Incrementing instructions, 36
Index register X, 11, 14, 35–37
Index register Y, 11, 14, 35–37
Indexed addressing, 80–85
  absolute, 81

indirect, 82–85
zero page, 80–81
Indirect addressing, 82–85
postindexed, 84–85
preindexed, 82–84
Inherent addressing, 75
INIT, 111
INSERT, 99, 199
Instruction-mnemonic, 74
Instruction set, 196–98
Interfacing to a printer, 168–72
Interrupt disable flag, 30, 33, 174, 177–78
Interrupts, 173–81
ACIA, 187
maskable, 174–75
nonmaskable, 176
software, 173, 176
Inverse video, 93–94, 97–99
I/O decoding, 106–107, 145–54
I/O SELECT, 141–43, 147–53, 157–58
I/O slots, 107
I/O STROBE, 141–43, 147–48, 157–58

Keyboard, 68–72
KEYIN, 103
KSWL, 155
KSWH, 155

LIFO stack, 57
Lines, (*See* Plotting)
LIST, 94, 199
Logical operations, 76
Low-resolution graphics, 106–17
colors, 111–12

Machine cycle, 62
Machine language, 1, 29
Mark, 183
Memory, 17
buffers, 118–21
change, 22, 199
decoding, 145–54
display, 19
map, 107
MESS, 94–99
Microcomputer, 1–2
Microprocessor, 1–5
Monostable multivibrator, 135–37
MOS Technology, Inc., 3
Motorola,
MC14443, 137, 139–40, 206–209
MC6821, 160, 210–20
MC6850, 182, 221–30
MC1488, 191–92
MC1489, 191–92
MC14411, 191–92

Negative flag, 30, 32, 55
Negative numbers, 41–44
Normal video, 93–94, 100
Number systems, 9–10

Octal numbers, 10
Offsets (See branching offsets)
One's complement, 42
Op-code, 28–30
Operand, 74
Operation code, (See Op-code)
OR, 76
Overflow, 44–46, 48
flag, 30, 32, 56

Parity, 183, 189
Peripheral boards, 153–59, 166–68, 191–92
slot independent, 155–57
Peripheral interface adapter (See PIA)
Peripheral I/O slots, 141–59
PIA, 145, 160–72, 177–78
control lines, 161–66
handshake mode, 165–66
on peripheral board, 166–68
pulse mode, 165
registers, 161–66
PLOT, 112–13, 116
Plotting:
lines, 114–16, 126–27
shapes, 127–33
spot, 112–13, 125–26
Position cursor, 20–21
PRBYTE, 105
PREAD, 137
PRHEX, 104–105
Printer, interfacing, 168–72
Printer output, 172, 199
Printing
byte, 105
characters, 92–93, 103–104
hex values, 104–105
message, 93–101
PRNTAX, 105
Program counter, 11, 29–30
Programmable read only memory, (See PROM)
PROM, 2, 145

Pull instructions, 38, 58–59
Pulse mode (See PIA)
Push instructions, 38, 58–59
Pushbutton inputs, 134–36, 139

QUIT, 201
RAM (Random access memory), 17, 106–107, 145, 147
  scratchpad for I/O slots, 152
Raster scan displays, 86–91
RDKEY, 103–104, 155
Read only memory (*See* ROM)
Real-time clock, 178–81
Register change, 13, 199
Registers
  ACIA, 184–91
  PIA, 161–66
  6502, 11–14, 27–38
Relative addressing, 79
REPT key, 20
Reset, 174
RETURN key, 13
Reverse video, (See inverse video)
Rockwell International, 3
ROM, 2, 106–107, 145, 147
ROT, 130, 132
Rotate instructions, 33–35
RS232, 191

SCALE, 130, 132
SCRN, 116
Screen display, 86–105
Screen soft switches
  (See soft switches)
Scrolling through memory, 20
Serial I/O, 182–93
SETCOL, 111, 116
SETGR, 110–111, 116
SETTXT, 111
Shape table, 127–33
Shifting instructions, 33–34
Single step, 28–30
Six-channel A/D converter, (See A/D converter)
Slot independent boards (See peripheral boards)
6502 microprocessor, 3–6
Soft switches, 106–11, 118–19
Sound (See Speaker)
Space, 183
Space bar, 20
Speaker, 65–68
Stack, 57–72, 174–75
Stack pointer, 11, 37–38, 58
Start bit, 182–83
Status flags, 30–32
Status register, 11, 14, 30–33, 39
Stop bit, 183
Strobe output, 134
Subroutines, 57, 59–62
Subtraction, 28, 31
Synertek, Inc., 3

Terminal, 191–93
Text mode, 108–111
Text window, 102
Three-state, (See Buffer)
Transfer block of bytes, 200
Transfer instructions, 36–37
TTL inputs (See Pushbutton inputs)
TUTOR monitor, 1–2
Two's complement, 41–43
TV RAM, 24, 88–94, 152

USR, 205

Video signal, 88
VLIN, 115–16

XDRAW, 127–28, 132–33

Z flag, (See Zero flag)
Zero flag, 30, 32, 51, 54
Zero page addressing, 77–78
  indexed, (See Indexed addressing)

Technical

Here's a specially designed book/software tutorial that makes it easy to learn assembly language programming and interfacing with the Apple II microcomputer.

Apple II-6502 Assembly Language Tutor includes a unique monitor program called TUTOR that makes learning assembly language programming easier than ever before. The TUTOR displays memory and register contents, allowing you to see the effect each assembly language program statement has on each location.

You'll also get a series of exercises that gives you hands-on experience in writing simple machine language programs and implementing various interfacing techniques. Practical and easy to use, *Apple II-6502 Assembly Language Tutor* is *the* book that provides you with the easiest, most effective way to develop assembly language programming and Apple II interfacing skills.

Contents include:

- the 6502 microprocessor
- computer memory
- the 6502 registers
- 6502 arithmetic
- branching instructions
- the stack and subroutines
- addressing modes
- displaying characters on the screen
- low-resolution graphics
- high-resolution graphics
- using the game I/O connector
- using the peripheral I/O slots
- the 6821 peripheral interface adapter (PIA)
- interrupts
- serial I/O—The ACIA

Richard Haskell, a professor of engineering and chairperson of the electrical and computer engineering group in the School of Engineering at Oakland University, Rochester, Michigan, has taught both introductory and advanced courses on microprocessors and has designed microprocessor-based systems for industrial applications. He holds a Ph.D. in electrical engineering from Rensselaer Polytechnic Institute and has written numerous articles and books, including four others in the Prentice-Hall computer series titled *Apple Basic, Atari Basic, Pet/CBM Basic,* and *TRS-80 Extended Color Basic.*

Cover design by Jeannette Jacobs
Cover illustration by John Brandes
PRENTICE-HALL, Inc.
Englewood Cliffs, New Jersey 07632

P

ISBN 0-13-9230-8